Reasoning about Regular Properties: A Comparative Study

Tomáš Fiedor, Lukáš Holík(✉), Martin Hruška,
Adam Rogalewicz, Juraj Síč, and Pavol Vargovčík

Brno University of Technology
{ifiedortom,holik,ihruska,rogalew,sicjuraj,ivargovcik}@fit.vutbr.cz

Abstract. Several new algorithms for deciding emptiness of Boolean combinations of regular languages and of languages of alternating automata have been proposed recently, especially in the context of analysing regular expressions and in string constraint solving. The new algorithms demonstrated a significant potential, but they have never been systematically compared, neither among each other nor with the state-of-the art implementations of existing (non)deterministic automata-based methods. In this paper, we provide such comparison as well as an overview of the existing algorithms and their implementations. We collect a diverse benchmark mostly originating in or related to practical problems from string constraint solving, analysing LTL properties, and regular model checking, and evaluate collected implementations on it. The results reveal the best tools and hint on what the best algorithms and implementation techniques are. Roughly, although some advanced algorithms are fast, such as antichain algorithms and reductions to IC3/PDR, they are not as overwhelmingly dominant as sometimes presented and there is no clear winner. The simplest NFA-based technology may sometimes be a better choice, depending on the problem source and the implementation style. We believe that our findings are relevant for development of automata techniques as well as for related fields such as string constraint solving.

1 Introduction

Efficient representation of regular properties of finite words has been the subject of research for a long time, with applications and results spanning much of the field of formal reasoning, including regular expression matching, verification, testing, modelling, or general decision procedures of logics. When regular properties are combined using Boolean and similar operations, interesting decision problems are PSPACE-complete. This includes the most essential problem of language emptiness (further just emptiness). The textbook approaches that use deterministic automata are plagued by state space explosion. Determinization and complementation is done by exponential subset construction and conjunction is quadratic. This motivated the research on efficient algorithms for non-deterministic and alternating finite automata (NFA and AFA, respectively).

Using nondeterminism and alternation, one can gain one or two levels of exponential savings in the size of automata, respectively. Alternation in context of automata was first studied in [24] and [18,53,38], and extensively in the context of automata over infinite words and temporal logics (e.g., [66,58,76,57]). It adds conjunctive branching to the disjunctive non-deterministic branching and allows to avoid the blow-up in the automata size completely. However, from the perspective of the worst case complexity, the gained succinctness is payed back by the PSPACE-completeness of language emptiness. Still, the more succinct the representation gives more opportunities for clever heuristics that combat the worst case complexity and work in practical cases, essentially by avoiding re-creation of the entire (non)deterministic representation.

Several very promising techniques and their implementations were proposed during the recent years. The latest advances in testing AFA emptiness appeared in the context of analysing combinations of regular expressions and in string solving. A group of these techniques is based on reducing AFA emptiness to a reachability in a Boolean transition systems and using existing implementations of model-checking algorithms, most notably of IC3/PDR [46,15], such as ABC [17], nuXmv [22], or IC3Ref [16], to solve it [28,27,47,80]. The most recent contribution from [73] extends the SMT-solver Z3 with symbolic derivatives, a generalisation of Antimirov derivatives of regular expressions. Z3 uses them to convert a combination of regular expressions into an alternating/Boolean automaton and on the fly tests its language emptiness through the classical de-alternation and a search for an accepting configuration.

Slightly older algorithm for testing equivalence of AFA (convertible to an emptiness test) is based on computing bisimulation up-to congruence [30]. It generalizes the original NFA-equivalence test of [11]. The congruence closure algorithms were preceded by the antichain algorithms that optimize the subset construction by the subsumption pruning [82,41], and by the first attempt to use the model checking algorithms, namely the algorithm Impact of [63], to emptiness of combinations of regular properties [40]. Lastly, the area of string constraint solving gave rise to a large variety of string constraint solvers. They approach combinations of regular properties through a spectrum of clever techniques based e.g. on automata, transformations to other types of constraints, reasoning on lengths of strings, Parikh images, etc. (e.g. Z3 [65,73], CVC4/5 [7,68], Z3Str4 [9], OSTRICH [25,26], Trau [5,4] to name a few).

These works demonstrate a significant promise, but they are presented in specific, often narrow contexts and under varying views on state of the art. Consequently, they have never been sufficiently compared against each other. Even comparisons against the most efficient implementations of the more standard techniques based on (non)deterministic automata is rare. String solvers were compared only against string solvers, advanced AFA-emptiness tests were compared only against the basic de-alternation. A somewhat interesting comparison was done only between NFA-antichain and up-to congruence-based language inclusion and equivalence test in [11] and in [39], and between the basic antichain based AFA emptiness and a version that uses abstract interpretation [41]. A number of works also take as their baseline implementations of automata or string solvers which, even though being respectable tools in their own right, are currently not the fastest solvers of combinations of regular properties in either category. On top of that, all the mentioned works on solving combinations of regular properties use only narrow benchmarks, often mutually exclusive.

Systematic comparisons of tools and algorithms on meaningful benchmarks is obviously needed to answer the questions ‘What to use?’ and ‘What to compare with?’, and generally for the field of reasoning about regular properties and automata to progress. We thus present a comparison of implementations of major algorithms. We compare the tools on a large benchmark of problems that we have collected from other works, from string constraint solving problems, analysis of regular expressions, regular model checking, and analysing LTL properties of systems. We believe that it is currently the most comprehensive benchmark in existence. Our main focus is on examples around string solving and analysis of regular expressions, which is also where the most of the recent developments has happened. These benchmarks mostly allow for a relatively simple representations of automata transition functions. Even though the alphabets in examples coming form this are large (e.g. UNICODE with up to 2^{32} symbols), the alphabet size can, in most cases, be reduced to few symbols by working with alphabet minterms (classes of indistinguishable symbols) instead of individual symbols. The issue of effective symbolic representation of transition relations with large alphabets then does not dominate the evaluation, although it would be critical in other application areas, such as deciding WS1S (monadic second-order logic of one successor) or linear integer arithmetic [20,81,44].

We have obtained results that paint the basic landscape of the available techniques and tools. They identify tools and approaches which are likely to work well and should be used as the baseline in comparisons. We also provide a relatively diverse and large benchmark to be used in comparisons. The results broadly confirm that the new algorithms represent a leap in efficiency compared to the technology of DFA and also make a reduction of a problem to language emptiness of alternating automaton an attractive option. On the other hand, they challenge some folklore knowledge and conclusions implied elsewhere. For instance, reductions to IC3/PDR, although yielding one of the fastest algorithm, are not as vastly superior as sometimes presented. Some practically relevant benchmark categories are best solved by a combination of an antichain algorithm with a SAT solver. Others, surprisingly many in fact, by a simple efficiency oriented implementation of basic algorithms for nondeterministic automata. Our results also underscore that there is no universal silver bullet. The particular kind of the problem, determined to a large degree by its source, is a decisive factor that should be taken into account when choosing and tuning a solver.

We will maintain and further grow the benchmark set, at GitHub [1], as well as the framework for the entire comparison, at [2], in order for it to be easily usable and extensible by others.

2 Preliminaries

A *(nondeterministic) finite automaton (NFA)* over Σ is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of *states*, Δ is a set of *transitions* of the form $q \text{-}\{a\}\text{→} r$ with $q, r \in Q$ and $a \in \Sigma$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A *run* of $\mathcal{A}$ over a word $w \in \Sigma^*$ is a sequence $p_0 \text{-}\{a_1\}\text{→} p_1 \text{-}\{a_2\}\text{→} \dots \text{-}\{a_n\}\text{→} p_n$ where for all $1 \leq i \leq n$, it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $w = a_1 \cdot a_2 \cdots a_n$, and either $p_{i-1} \text{-}\{a_i\}\text{→} p_i \in \Delta$ or $p_{i-1} = p_i$, $a_i = \epsilon$. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the *language* $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all words for which $\mathcal{A}$ has an accepting run.

The automaton is *deterministic (DFA)* if for every state q and symbol a, Δ has at most one transition $q \text{-}\{a\}\text{→} r$. Any NFA can be determinized by the *subset construction*, which creates the DFA $A' = (2^Q, \Delta', \{I\}, \{S \mid S \cap F \neq \emptyset\})$ where $S \text{-}\{a\}\text{→} S' \in \Delta'$ iff $S' = \{s' \mid s \in S \wedge s \text{-}\{a\}\text{→} s' \in \Delta\}$. The basic automata constructions implementing Boolean operations with languages are intersection, $\mathcal{A} \cap \mathcal{A}' = (Q \times Q', \Delta^{\times}, I \times I', F \times F')$ where

$(q,q')\text{-}\{a\}\!\!\rightarrow(r,r') \in \Delta^\times$ iff $q\text{-}\{a\}\!\!\rightarrow r \in \Delta$ and $q'\text{-}\{a\}\!\!\rightarrow r' \in \Delta'$, non-deterministic union $\mathcal{A} \cup \mathcal{A}' = (Q \cup Q', \Delta \cup \Delta', I \cup I', F \cup F')$, deterministic union by product which is the same as $\cap$ up to that the final states are $F \times Q \cup Q \times F$, and complementation which consists of determinization and complementing the final states.

Alternating automata. An *alternating finite automaton (AFA)* in the most general form would be a tuple $\mathcal{M} = (\Sigma, \mathbb{P}, Q, \delta, I, F)$ where, when denoting $\mathbb{B}(X)$ the Boolean predicate formulae over variables X: 1) Σ is a finite alphabet; 2) $\mathbb{P}$ is a set of unary *symbol predicates* with a free variable α; 3) Q is a finite set of *states*; 4) $\delta : Q \to \mathbb{B}(Q \cup \mathbb{P})$ is a *transition function* where states of Q have only positive occurrences 5) $I \in \mathbb{B}(Q)$ is a positive *initial condition*; and 6) $F \in \mathbb{B}(Q)$ is a negative *final/accepting condition.* [1]

It can be interpreted as the *forward NFA* $A^{\mathsf{f}} = (\Sigma, \mathcal{P}(Q), \Delta^{\mathsf{f}}, I', F')$ with states $c \subseteq Q$ called *configurations* of A. Assume many sorted interpretation of formulae over variables Q of the type Boolean (values 0 and 1) and the variable α of the type Σ. A set of states $c \subseteq Q$ is understood as an assignment $Q \to \{0,1\}$ in which $c(q) = 1$ corresponds to $q \in c$. A pair (c,a), $a \in \Sigma$ is understood as the same assignment extended with $\alpha \mapsto a$. The satisfaction relation $\models$ between a formula and a configuration c or a pair (c,a) is defined as usual. The transition relation Δ^{f} then contains a transition $c\text{-}\{a\}\!\!\rightarrow c'$ iff $(c',a) \models \bigwedge_{q \in c} \Delta(q)$, and I' and F' are the sets of configurations that satisfy I and F, respectively. It is common to define Δ^{f} to contain only the smallest transitions, that is, for a given c and a, only the transitions $c\text{-}\{a\}\!\!\rightarrow c'$ with the $\subseteq$-minimal target c' are in Δ.[2] The language of A, $L(A)$, is the language of A^{f}.

The AFA can equivalently be interpreted as the *backward NFA*, the automaton $A^{\mathsf{b}} = (\Sigma, \mathcal{P}(Q), \Delta^{\mathsf{b}}, I', F')$ where $c\text{-}\{a\}\!\!\rightarrow c' \in \Delta^{\mathsf{b}}$ if $(c,a) \models \Delta(q)$ for each $q \in c$. Here it is enough to take, for a given c' and a, only the transition with the $\subseteq$-largest source c[3] (this makes the transition relation backward deterministic).

Boolean automata. Alternating automata may be extended to Boolean finite automata (BFA) by allowing any Boolean combination in the initial, final, and transition formulae (states in the initial and transition formulae may occur negatively, states in the final formula may occur positively). Note that the extension of AFA to BFA is not dramatic, as a BFA is easily encoded as an AFA with only double the size, by the following steps: 1) for each $q \in Q$, add state $\bar{q}$ with $\Delta(\bar{q}) = \neg\Delta(q)$, 2) transform all formulas in I, F, Δ to DNF, 3) replace all literals $\neg q$ by $\bar{q}$ in Δ and I and replace literals q by $\neg\bar{q}$ in F.

Restricted forms of AFA transition relation. The general form of AFA, as defined above, is the most succinct. It provides space for most optimizations, such as in [77]. Automata in this form are generated from LTL conversions of [34] used in [30,77]. On the other hand, only a small subset of algorithms and tools support AFA in this most liberal form. A common restriction (used e.g. in [30]) is to separate symbols from states in the transition formulae, that is, having $\Delta(q)$ in the form $\varphi \wedge \psi$ with $\varphi \in \mathbb{B}(\mathbb{P}), \psi \in \mathbb{B}(Q)$.

[1] This is not a most standard definition of AFA but it allows us to later cover and categorize their common syntactic variants. See e.g. [41,18,57] for more standard definitions.

[2] A state in a configuration is understood as a constraint. The less constraints, the more can be accepted from the configuration. Transitions to more constrained configurations are useless.

[3] Going backward, larger configurations are more permissive. Transitions from the same target with smaller configurations are useless.

We call such AFA *separated*. The transition relation can then be seen as a function $Q \to \mathbb{B}(\mathbb{P}) \times \mathbb{B}(Q)$. Separated AFA are often considered with the state formula ψ in the disjunctive normal form (e.g. in [36,41]), which we call the *DNF form*, and Δ then may be seen as a set of transitions of the form $q \text{-}\{\varphi\}\!\!\rightarrow c$ where $\bigwedge c$ is a (positive) clause of ψ.

The decision problems. We will concentrate on two decision problems:
(1) *AFA emptiness* asks whether the language of the given AFA is empty.
(2) *Emptiness of Boolean combinations of regular properties* (*BRE*), asks whether a Boolean combination of regular languages, given as automata or regular expressions, is empty (languages can be combined with $\cap$, $\cup$, and complement wrt. Σ^*, which also covers testing inclusion and equivalence[4]).

3 Existing Algorithms and Tools

In this section, we will overview the existing approaches and tools implementing AFA and BRE emptiness.

3.1 Representation of Automata Transition Relations

In the simplest form, a predicate on a automata transition represents a single letter from the alphabet. This is called an *explicit transition*. Explicit automata are simple, allow for low level optimizations, and implementation of complex algorithms for them is manageable (such as advanced algorithms for computing simulations [70,50,23]). The technique of a-priori mintermization, that replaces the alphabet by the alphabet of minterms, classes of indistinguishable symbols, makes explicit automata usable also when alphabets are large. However, when the number of minterms tends to explode, explicit automata do not scale.

Various implementations of automata have been using transition predicates implemented as BDDs, Boolean formulae, formulae over SMT-theory of bit-vectors, intervals of numbers, etc. This has been systematized in the works on *symbolic automata* [79,33,31], where the symbol predicates may be taken from any effective Boolean algebra (and the automata are in the separated form). Even more compact than symbolic automata are representations of the transition relation used in the WS1S solver MONA or in some of the implementations of AFA, which in a way drop the restriction to the separated form. We will discuss the concrete implementations below.

3.2 (Non)deterministic Finite Automata

The baseline approach to solve BRE is to use DFA or NFA. Boolean operations are implemented as the classical construction listed in Section 2. Automata may be kept deterministic, or they are kept non-deterministic whenever possible and determinized only before complementing. An important ingredient of achieving efficiency is usually to minimize automata at least once every few operations (important e.g. in applications such as regular model checking [12] or some approaches to string solving [10,25,4]). The deterministic approaches construct the minimal DFA by the Hopcroft, Moore, Brzozowski, or the Huffman algorithm [52,64,19,54], the non-deterministic approach may use simulation [70,23,45,55,50] or bisimulation [75,69,48] based reduction methods.

[4] $L' \subseteq L$ is emptiness of $L' \cap \overline{L}$ and equivalence is emptiness of $(L' \cap \overline{L}) \cup (\overline{L'} \cap L)$.

Simulation reduces significantly more but is much costlier. DFA/NFA are implemented in many libraries. Here we select a representative sample.

First, ENFA is the simplest tool, our own implementation of NFA, which was originally meant to play the role of a baseline. It uses explicit automata with mintermization. It is implemented in C++, with efficiency in mind, but with no extensive optimizations (roughly, transitions from a state stored in a two layered data structure, the first layer divided and ordered by symbols, and the second layer ordered by the target state). It uses an off the shelf implementation of one of the newest generation algorithms for computing simulation [70,23,50] (that achieve good efficiency through a usage of the partition-relation data structure) taken from VATA tree automata library [59] (implementing namely [50]).[5]

The BRICS automata library [67] is often considered a baseline in comparisons [67]. It uses primarily deterministic automata and transition relation represented symbolically using character ranges. It is written in Java and relatively optimized.

The AUTOMATA library [78], made in C#, implements symbolic NFA/DFA parametrized by an effective Boolean algebra. We use it with the default algebra of BDDs. AUTOMATA has been long developed and has accumulated many optimizations and novel techniques for handling symbolic automata (e.g., optimized minimization [32]).

MONA [44], written in C, is the most influential and optimized implementation of deterministic automata. It specialises in deciding WS1S formulae, which besides Boolean combinations includes also quantification. The decision procedure generates DFA with complex transition relations over large alphabets of bit-vectors. For this purpose, MONA uses a compact representation of the transition relation: a single MTBDD for all transitions originating in a state, with the target states in its leaves. MONA can represent only a DFA, hence it always implicitly determinizes.

VATA [59], written in C++, is a library implementing non-deterministic tree automata. As NFA are a special case of tree automata, we can use it as an implementation of the basic constructions for explicit NFA. It is relatively optimized. We include it into the comparison for its fast implementation of the antichain inclusion checking [12,49], which for NFA boils down to the inclusion check of [36].

3.3 Alternating Automata

De-alternation. The basic approach to AFA emptiness is *de-alternation*, transformation to an NFA, either the forward A^{f} or the backward A^{b}, followed by testing the emptiness of the resulting NFA. Both NFAs are constructed by a variation on the NFA subset construction. We are not aware of any tool using pure de-alternation, and we believe that it would not be competitive. The forward algorithm is however the basis of [73] used in Z3 where it is run on the fly with a novel symbolic derivative construction (discussed also in the paragraph on string constraint solvers).

Interpolation based abstraction refinement. Attempts to harness model checking algorithms to AFA emptiness appeared in the context of string solving and processing of regular expressions. To our best knowledge, the earliest attempt was [40], where conjunctions of regular constraints were solved using the interpolation-based algorithm of [62]. The interpolation-based abstraction refinement, namely the algorithm Impact of

[5] In our experiment, simulation is only used after parsing and has minimal overall impact.

[63], was also used in [56]. This work concentrated on more general problem, solving emptiness of AFA over data words with an infinite data domain (that can relate past and current values of data variables). Their tool JALTIMPACT [3] (in Java), that we include into our comparison, can be run on our benchmark too.

Reduction to reachability and IC3/PDR. The work of [80] presented the first translation of string constraints (mostly BRE) into reachability in a Boolean transition system (circuit) that was then solved by the model checker nuXmv [22]. This was de facto the first reduction of AFA emptiness to reachability in a Boolean transition system (BTS).

Let us briefly overview the basic principle of the reduction. The *forward BTS* for an AFA A has configurations that are Boolean assignments to Q, initial and final configurations satisfy I and F, respectively, and transitions are given by the formula $\Phi^{\mathsf{f}}_{\Delta} : \bigwedge_{q\in Q} q \to [\Delta(q)]'$. Here we use $[\varphi]'$ to denote the formula obtained from φ by substituting every state q by its primed version q', and we will also denote by $[c]'$ the primed version $\{q' \mid q \in c\}$ of a configuration c. A *successor* of a configuration c is any configuration $\bar{c}$ such that $[\bar{c}]'$ satisfies $\exists Q \exists \alpha\, \Phi^{\mathsf{f}}_{\Delta} \wedge \bigwedge_{q\in C} q$ (the symbol variable alpha is of the bit-vector sort). *Reachability* is then the transitive and reflexive closure of the successor relation and the *reachability problem* asks whether a final configuration is reachable from an initial one. It is the case if and only if A is not empty. The forward reduction has been used in [80]. Alternatively, the *backward BTS* for A has the initial configurations satisfying F, final configurations satisfying I, and the successor relation given by the formula $\Phi^{\mathsf{b}}_{\Delta} : \bigwedge_{q\in Q} q' \to \Delta(q)$.

The work [28] applied IC3/PDR [46,15], implemented in IC3Ref [16], together with the backward BTS reduction to solve emptiness of BRE and obtained very encouraging results. The implementation used in [28], called Qzy, is, however, proprietary and not publicly available. Similar approach was taken by [47], where a string constraint was translated to a multi-tape AFA and then to a BTS by the forward translation, and given to IC3/PDR to solve through tools nuXmv [22] or ABC [17]. Results of [77] seem to indicate that the backward translation is better and the same is suggested by the comparison in [28,27] in which the string solver Sloth [47], based on the forward reduction, was much slower than Qzy, based on the backward reduction. In this comparison, we include our own C++ implementation BWIC3 of the backward reduction based on the model checker ABC.

Antichains. Antichain algorithms presented in [82] were the first breakthrough in solving BRE. They use subsumption relations between the states of the automata constructed by variations of the subset construction to prune the constructions. They were used to test language universality and inclusion of NFAs and AFA emptiness. The AFA emptiness namely is based on an on-the-fly search for an accepting state of the A^{f} or for an initial state of the A^{b}. Subsumption prunes discovered states that are larger (smaller for the backward algorithm) than others.

The antichain algorithms were enhanced and generalized in a number of works, e.g. with a more aggressive pruning by the simulation-based subsumption [6,36], or by counterexamples guided abstraction refinement in [41]. In this comparison, we include the NFA inclusion check implemented in the VATA tree automata library [59]. We also experimented with a student-made implementation of the antichain AFA emptiness check of [41] that uses abstraction refinement (the original implementation is no longer maintained and we were not able to run it). However, not being able to achieve a competitive performance, we excluded it from the comparison. One reason of the poor performance may be that simplest form of AFA, explicit DNF form (used in the original version [41]), might be too inefficient and costly to construct in our examples, partly due to a large number of minterms induced by the AFA emptiness benchmark.

We implemented (in C++) the antichain AFA emptiness test of [36] that integrates tightly with a SAT solver to handle the general form of AFA with large alphabets. We will refer to it as ANTISAT. We will briefly explain its principle. It essentially implements the reachability test for the backward BTS discussed in the previous paragraph. A configuration c is represented by the conjunction $\phi_c = \bigwedge_{q \in Q \setminus c} \neg q$. Note that ϕ_c is satisfied by the downward closure of c, which are all configurations included in (subsumed by) c. To compute predecessors of configurations represented by ϕ_c, the SAT solver (namely MiniSAT [37]) is called on the formula $\Phi : \Psi^{\mathsf{b}}_{\Delta} \wedge \phi_c \wedge \psi_{\mathsf{Ach}}$. Here, ψ_{Ach} excludes all already discovered configurations from the solution. It is a conjunction of clauses $\overline{\phi_c} : \bigvee_{q \in Q \setminus c} q$ for every previously discovered configuration c. The SAT solver discovers a satisfying assignment e, which is turned into a new configuration $c' = Q \cap e$ (that is, the values of the symbol bits constituting the bit-vector α are omitted from e). Unless c' is initial, it is queued for further predecessor computation and is immediately added to ϕ_{Ach} through the interface of incremental SAT solving as the clause $\overline{\phi_{c'}}$. Finally, only maximal predecessors of c are of interest, as the non-maximal ones are subsumed by them. We enforce the maximality of c through working directly with the internal SAT solver structures: at decision points, the SAT solver is forced to give priority to decisions that assign 1 to state variables.

Bisimulation up-to congruence. A later class of algorithms, here refered to as *up-to algorithms*, checks equivalence as a bisimulation between configurations of AFA, and utilises the up-to congruence technique to prune the search space. The first algorithm on NFA equivalence [11] was extended to alternating automata emptiness check in [30]. These algorithms are close to antichains. As shown in [11], the pruning potential of the up-to techniques is in theory the same or larger than that of antichain. A disadvantage of the up-to congruence technique is the need for expensive evaluation of congruence closures. The more extensive experiments of [39] shows antichain algorithms as faster, with an exception of randomly generated automata with small alphabets and very dense transition relations. We include into the comparison the Java implementation of the AFA-emptiness of [30] (emptiness reduces to equivalence with a trivial empty AFA), that we refer to as BISIM. The other implementations of up-to algorithms we are aware of, from [39] and [11], are single-purpose programs that decide equivalence of two NFAs, hence we would be able to run them on a very small fraction of our benchmark only.

3.4 String Constraints Solvers

There are dozens of string constraint solvers that implement, to a various degree, a support for deciding combinations of regular properties. String languages are rich and BRE are not the absolute priority of the solvers, hence they perform on them generally worse than specialised tools. However, string solvers implement a wide scale of unique techniques and pragmatic heuristics that may work in specific instances. Representatives of the solvers with the most mature implementations (also used in most comparisons in the literature) are Z3 [65,73] and CVC5 [68,7]. CVC5 solves BRE mostly through rewriting rules. Recently [73] extended Z3 with an approach based on the Antimirov derivative automata construction generalised to symbolic automata and extended regular expressions. Essentially, the construction produces a symbolic AFA/BFA and checks its emptiness on the fly while running the forward de-alternation. As shown in [73], it is significantly more efficient in solving BRE than other SMT solvers (including CVC5).

3.5 Other Approaches and Tools

Although we believe that we have collected a representative subset of existing algorithms and tools, we have not collected all interesting specimens. Some were not available, some were difficult to run or prepare the inputs for, some seemed covered by experimentation in other works. Including these tools and algorithms into the comparison could still be interesting and we leave it for the future work (we plan to keep extending the tool base as well as the benchmark set). Namely, the tool DPRLE [51], used in the comparison in [28], seemed to be mostly outperformed by the IC3/PDR approach implemented in Qzy, however, not absolutely consistently. The implementation of NFA antichain and up-to congruence techniques used in [39] seems efficient, with its NFA antichain inclusion twice as fast as that of VATA. The up-to congruence NFA equivalence checking of [11] could be fast too ([11] and [39] report somewhat conflicting results). There are numerous NFA/DFA libraries, e.g. the C alternative of BRICS [61] or the Java implementation of symbolic NFA of [29]. ALASKA [35] might contain interesting implementations of antichain algorithms but is no longer maintained and available. Our comparison is missing a basic implementation of antichain-powered de-alternation for explicit AFA in the DNF form, which, if not overwhelmed by a large number of minterms, could reach a good performance through simple fast data structures, similarly to our ENFA.

4 Benchmarks

We collected as comprehensive benchmark as possible, harvesting examples used in previous works as well as generating some of our own. It is available together with the whole experiment from [2] and at GitHub [1] (we plan to maintain and grow the benchmark and welcome contributors).

Our main focus of the current benchmark set is the areas where the most of the development in solving AFA and BRE emptiness happened recently, which is string constraint solving and analysis of regular expressions used in analysing and filtering texts. Atomic regular properties are here mostly given in the form of regular expressions over UNICODE character classes. The alphabet is large but the number of minterms is mostly small or moderate. This is true also for our examples from regular model checking. Symbolic handling of complex transition relations over large alphabets is thus not absolutely crucial and the experiment can stay focused on the main algorithms for emptiness check. For that reason, we do not include benchmarks from solving WS1S [21], the primary target of MONA, or Presburger arithmetic with automata [13,81], where the techniques of handling symbolic alphabet are indispensable. Techniques specialising at this kind of problems would deserve their own study. Our benchmarks where the symbolic alphabet representation is still rather important are AFA coming from (combinations of) LTL properties, with alphabets of sets of atomic propositions, and from translations of string constraint problems to AFA with complex multi-track alphabets.[6]

Boolean combinations of regular expressions. This group of BRE contains benchmarks on which we can run all tools, including those based on NFA and DFA. They have small to moderate numbers of minterms (about 30 in average, at most over a hundred).

b-smt contains 330 string constraints from the Norn and SyGuS-qgen, collected in SMT-LIB benchmark [8], that fall in BRE. These were also used to compare SMT-solvers in [73].

b-hand-made has 56 difficult handwritten problems from [73] containing membership in regular expressions extended with intersection and complement. They encode (1) date and password problems, (2) problems where Boolean operations interact with concatenation and iteration, and (3) problems with exponential determinization.

b-armc-incl contains 171 language inclusion problems from runs of abstract regular model checking tools (verification of the bakery algorithm, bubble sort, and a producer-consumer system) of [12]. These examples were used also in [39,11].

b-regex contains 500 problems, obtained analogously as in [30,77], of the form $r_1 \wedge r_2 \wedge r_3 \wedge r_4 = r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$, where each r_i is one of the 75 regexes[7] from RegExLib [71] selected so that $r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$ is not empty. This benchmark is inspired by spam filtering, where we want to test whether a new filter r_5 adds anything to existing filters. We transformed this problem into the inclusion $r_5 \subseteq r_1 \wedge r_2 \wedge r_3 \wedge r_4$, and kept the original form for Bisim which expects an equivalence.

b-param has 8 parametric problems. Four are from [40]:
(1) `[a-c]a[a-c]`$\{n+1\} \cap$ `[a-c]a[a-c]`$\{n\}$ (long strings),
(2) $\bigcap_{i=1}^{n}$ `([0-1]`$\{i-1\}$`0[0-1]`$\{n-1\}$`0[0-1]`$\{n-i\}\alpha_i$`)|([0-1]`$\{i-1\}$`1[0-1]`$\{n-1\}$`1[0-1]`$\{n-i\}\alpha_i$`)` (exponential branching),
(3) $\bigcap_{i=1}^{n}$ `.*(.`$\{p_{10+i}\}$`)+`α_i (exponential paths 1), and
(4) $\bigcap_{i=1}^{n}$ `.+`α_i`0(.`$\{p_{10+i}\}$`)+` (exponential paths 2), where $\alpha_1, \ldots, \alpha_n$ are disjoint character classes and p_j is the j-th prime number. Another four are from [28]:
(5) `^.[01]*.1.[01]{n}.$` $\setminus$ `^.[01]*.0.[01]{n-1}.$` (sat. difference),
(6) `^.[01]*.1.1.[01]{n}.$` $\setminus$ `^.[01]*.0.[01]{n+1}.$` (unsat. difference),
(7) `^.[01]*.1.[01]{n}.$` $\cap$ `^.[01]*.0.[01]{n-1}.$` (sat. intersection) and
(8) `^.[01]*.1.[01]{n}.$` $\cap$ `^.[01]*.0.[01]{n}.$` (unsat. intersection). For (1) we chose $n \in \{50, 100, \ldots, 500\}$, for (2)-(4) we chose $n \in \{2, 3, \ldots, 60\}$ and for (5)-(8) we chose $n \in \{50, 100, \ldots, 1000\}$.

[6] We did not attempt to generate purely random problems. First, purely random automata generated e.g. by [74] seem to have different characteristics than automata coming from practical problems (e.g. in [12,39]). Second, although generating random NFA is possible with a generator controlled by three simple parameters which give a manageable parameter-value space covering all NFA, it is not clear how to similarly generate random AFA or BRE. On the other hand, we do include a benchmark based on randomly generated LTL formulae, which we consider relatively close to realistic LTL specifications.

[7] `https://github.com/lorisdanto/symbolicautomata/blob/master/benchmarks/src/main/java/regexconverter/pattern%4075.txt`

AFA Benchmark. The second group of examples contains AFA not easily convertible to BRE. Here we can run only tools that handle general AFA emptiness. Some of these benchmarks also have large sets of minterms (easily reaching to thousands) and complex formulae in the AFA transition function, hence converting them to restricted forms such such as separated DNF or explicit may be very costly. This also seems to be the main reason for which our implementation of [41] could not compete.

a-ltlf-patterns comes from transformation of linear temporal logic formulae over finite traces (LTL_f) to AFA [34]. The 1699 formulae are from [60][8] and they represent common LTL_f patterns which can be divided into two groups: (1) 7 parametric patterns (100 each) and (2) randomly generated conjunctions of simpler LTL_f patterns (999 formulae).

a-ltl-rand contains 300 LTL_f formulae obtained with the random generator of [77]. The generator traverses the syntactic tree of the LTL grammar, and is controlled by the number of variables, probabilities of connectives, maximum depth, and average depth. We have set the parameters empirically in a way likely to generate examples difficult for the compared solvers (the formulae have 6 atomic propositions and maximum depth 16).

a-ltl-param has a pair of hand-made parametric LTL_f formulae (160 formulae each) used in [30,77]: *Lift* [43] describes a simple lift operating on a parametric number of floors and *Counter* [72] describes a counter incremented modulo the parameter.

a-ltlf-spec [60] contains 62 LTL_f formulae that specify realistic systems, used by Boeing [14] and NASA [42]. The formulae represent specifications used for designing Boeing AIR 6110 wheel-braking system and for designing NASA NextGen air traffic control (ATC) system.

a-sloth 4062 AFA emptiness problems to which the string solver Sloth reduced string constraints [47]. The AFA have complex multi-track transitions encoding Boolean operations and transductions, and a special kind of synchronization of traces requiring complex initial and final conditions.

a-noodler 13840 AFA emptiness problems that correspond to certain sub-problems solved within the string solver Noodler in [10]. The AFA were created similarly as those of a-sloth, but encode a different particular set of operations over different input automata.

5 The Comparison

We ran our experiments on Debian GNU/Linux 11, with Intel Core 3.4GHz processor, 8 CPU cores, and 20 GB RAM. All experiments were run with the timeout of 60 seconds (increasing the timeout did not have a significant impact). Additional details as well as the virtual machine with the entire benchmark are available at [2].

Benchmarking infrastructure. The initial difficulty is that the tools expect different input formats and forms of automata and the benchmarks come in different formats as well. We converted all benchmarks to our internal AFA format, from which we generated formats supported by the AFA handling tools JALTIMPACT, BWIC3, ANTISAT,

[8] `https://drive.google.com/file/d/1eOYGvm3C8sQ-9iyfZ8qx42K54hgrFNTC`

and Bisim, or we extend the tools with a parser. The BRE benchmarks come from various sources. We first convert them into a master file which specifies the Boolean combination of atomic NFA, each atomic NFA stored in a separate file. The SMT-lib format is generated for Z3 and CVC5. In the case of b-hand-made, b-param, and b-smt, the atomic automata are translated from regular expressions using the parser of Brics, while in the case of b-regex, where the regexes contain features not supported by Brics, we use the parser from Bisim. b-smt and b-hand-made requires first translating from SMT-lib to a regular expression. In the case of b-armc-incl, the atomic automata come directly as NFAs, and are converted into formats of the individual BRE solvers (we again wrote parsers for some of the solvers), and to our AFA format for the AFA solvers. Every BRE solver was extended by an interpreter of the master file that reads the NFA/DFA from the generated solver-specific files (except the SMT solvers, which read SMT-lib). We note that due to some difficulties with internal structures, we currently cannot run Brics on b-armc-incl, and due to the lack of a converter from complex regular expressions and from pure NFA to the SMT format, we do not run Z3 and CVC5 on b-regex and on b-armc-incl.

Measured data. We will present the results obtained with BRE (where we run all the tools) and with AFA emptiness (where we run bwIC3, Antisat, Bisim, and JaltImpact) separately. We also separate the results on examples from applications from results on parametric hand-made examples.

Table 1 summarizes the statistics from evaluating the benchmarks. The table lists: (i) the average time, (ii) the median time, and (iii) the number of timeouts and number of errors (mostly, a tool ran out of the memory, made a bad alloc or ran into a segmentation fault). A few errors, e.g. in CVC5 or Bisim, were due to the unsupported features in the inputs. The tools' performance is then visualised on cactus plots in Fig. 1. For each tool, the plot shows the progress of the tool on each benchmark: the y axis is the cumulative time taken on the benchmark, with the individual examples on the x axis ordered by the runtime taken by the tool. Timeouts are omitted. In the appendix, we also show a set of scatter-plots that compare for every benchmark the three best performing tools.

Finally, we compared the tools on the parametric benchmarks a-ltl-param and b-param. We illustrate the results in Fig. 2. Each graph shows the times for the increasing value of the specific parameter on the x axis.

5.1 Discussion

Based on the measurements, we make several observations.

Firstly, the tool which combines universality (it can be run on AFA as well as on BRE emptiness) with the most consistent good performance is bwIC3. It dominates most of the AFA emptiness benchmark, shows great or a very good performance on the BRE benchmark, and often stands out on the parametric examples. Moreover, the measurements reported in [28] suggest that the backward BTS reduction has even more potential. This is visible namely from the comparison of our results on the parametric benchmarks diff-sat, diff-unsat, inter-sat, and inter-unsat. Our implementation matched the result of [28] on diff-sat and partially on inter-sat, saw a worse trend on diff-unsat and much worse trend on inter-unsat. A likely culprit is a different underlying model-checker, ABC [17] in our implementation versus IC3Ref [16] in [28]. However, IC3Ref

Table 1. Summary of AFA and BRE benchmarks. Table lists (i) the average, (ii) the median, and (iii) the number of timeouts and errors (in brackets). Winners are highlighted in bold.

	a-ltl-rand (300)			a-ltl-spec (62)			a-ltlf-patterns (1 699)			a-noodler (13 840)			a-sloth (4 062)			a-ltl-param (320)		
bwIC3	**0.1**	**0.1**	**0**	0.1	0.1	0	**0.1**	**0.1**	**0**	**0.1**	**0.1**	**3**	**1.3**	**0.1**	**34**	**25.4**	**0.6**	**134**
Bisim	4.4	1.0	8	32.9	60.0	32	37.0	60.0	1013	31.6	26.4	6644(8)	17.5	1.5	1087(10)	58.2	60.0	308
JaltImpact	7.9	2.3	12	2.4	1.4	0(1)	4.0	2.8	0	3.8	1.8	186	24.1	15.4	958	47.0	60.0	205
Antisat	18.3	0.1	84	**0.0**	**0.0**	**0**	31.0	60.0	868	0.4	0.0	57	14.9	0.0	991	58.3	60.0	310

	b-armc-incl (171)			b-hand-made (56)			b-regex (500)			b-smt (330)			b-param (267)		
bwIC3	5.2	1.1	1	0.4	0.1	0	**0.2**	**0.1**	**0**	0.1	0.1	0	44.9	60.0	191
Bisim	28.5	9.5	72	11.2	1.0	8	3.8	1.3	15	2.5	2.5	0	55.4	60.0	240
Brics		-		3.9	0.4	3	5.8	0.8	40	0.3	0.3	0	52.7	60.0	228
CVC5		-		27.4	0.8	10(15)		-		0.8	0.2	1	48.6	60.0	208
Automata	3.5	0.4	9	0.2	0.2	0	0.2	0.2	0	0.2	0.2	0	46.3	60.0	161(42)
JaltImpact	30.9	24.6	63	11.1	3.6	5	12.2	2.4	48	3.5	3.5	0	57.8	60.0	252
Antisat	42.8	60.0	118	1.4	0.0	1	9.3	1.4	45	0.0	0.0	0	39.0	60.0	147
Mona	28.5	44.1	43	27.3	0.1	22(3)	41.0	60.0	15(298)	1.5	0.0	8	44.9	60.0	25(169)
eNfa	**1.9**	**0.8**	**0**	**0.1**	**0.0**	**0**	**0.2**	**0.1**	**0**	**0.0**	**0.0**	**0**	44.6	60.0	143(51)
VATA	2.6	3.4	0	**0.1**	**0.0**	**0**	2.1	0.2	10(1)	**0.0**	**0.0**	**0**	37.8	60.0	155(1)
Z3		-		3.9	0.0	2		-		0.4	0.0	2	**32.0**	**48.1**	**129**

was not used out of the box in [28], harnessing it efficiently for problems of our king is not entirely trivial.

Secondly, the results on application related BRE (all BRE except the parametric examples in b-param) quite surprisingly favour the tools based mostly on relatively basic NFA algorithms. The overall best is the simplest tool of all, our implementation eNfa of basic NFA constructions. Close to the performance of eNfa is VATA, which uses the antichain inclusion checking on b-armc-incl and b-regex (the fact that explicit complementation of eNfa is faster than the antichain of VATA suggests that the inclusion benchmarks are not particularly hard). VATA specialises to the more general tree automata, which probably causes unnecessary overhead. Automata also performs well. It uses slightly more advanced algorithms than eNfa (such as lazy evaluation of difference, though, without antichain pruning). Its symbolic representation of transition functions with BDDs probably does not provide much advantage here. This result challenges the view that translating complex problems, arising for instance in string constraint solving, into AFA in order to use the sophisticated machinery of AFA solvers is an obvious silver bullet. Organizing the computation into smaller NFA operations, where, moreover, partial results can be minimized and re-used, and a simpler and hence more flexible NFA technology is used, might be a better strategy (this seems to work very well for instance in our recent prototype string constraint solver [10]).

Our AFA emptiness test Antisat based on the antichain algorithm and a SAT solver has an interesting performance. As can be seen on the cactus plots, besides its absolute domination on a-ltlf-spec, it is significantly faster than other tools on a large portion of the other AFA emptiness benchmark, but struggles on the rest. The examples where it dominates are often automata with the structure resembling a lasso (or several lassos) with a long handle. The other implementation of an antichain algorithm, NFA/NTA inclusion in VATA, also shows a good performance. This together points on the overall strength of antichain algorithms.

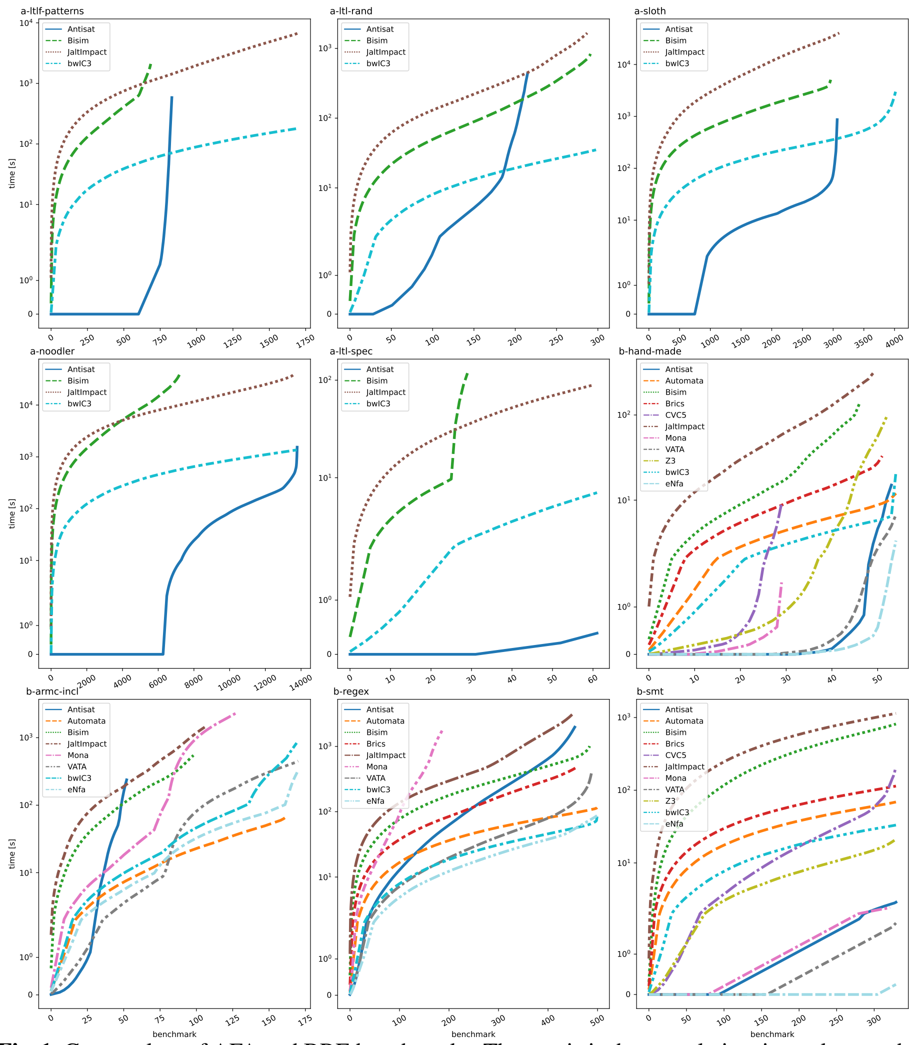

Fig. 1. Cactus plots of AFA and BRE benchmarks. The y axis is the cumulative time taken on the benchmark in logarithmic scale, benchmark on the x axis are ordered by the runtime of each tool.

The SMT string constraint solvers are not among the best in the benchmark related to practical applications, but are competitive (especially Z3), and win on some parametric cases. This may be due to that various heuristics unique to SMT solvers, especially rewriting that reduces one type of a constraint to another, kicks in. For instance, Z3 seems to solve exppaths1 with a help of rewriting to the sub-string constraint in the theory of sequences. In general, the measurements on parametric examples underscore the fact that no algorithm is universally the best and their relative performance may vary drastically depending on the kind of an input.

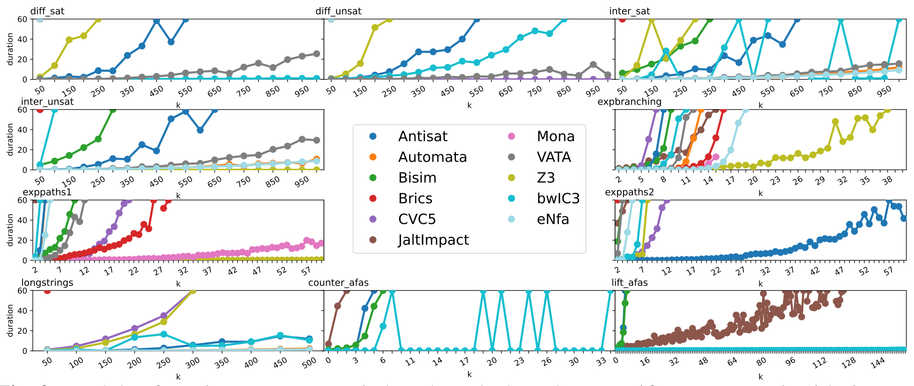

Fig. 2. Models of runtime on parametric benchmarks based on specific parameter k with timeout 60s. The sawtooths represent the tool failed on the benchmark for some k while solving benchmarks for $k-1$ and $k+1$. For brevity, we draw the models only until they start continually failing.

Although the mediocre performance of the other tools can be partially explained by their focus on a different kind of a problem or a dated underlying technology, and each of them is respectable in its own right, a point can be made against relying on them as a baseline in comparisons of tools for solving our kind of problem. MONA, optimized for a different settings (complex alphabets of bit-vectors with many minterms), is held back by the implicit determinization, and, in our case, probably by the overhead of the symbolic representation. It also frequently runs out of the 32-bit address space for BDD nodes. Similarly for BRICS, which also always determinizes. The low performance of BISIM is surprising relative to the good results of the up-to algorithms reported in [11,30]. It is more consistent with [39] where up-to algorithms were not wining against antichains on the more practical examples. Our results however do not directly contradict the results of [30] itself, since it does not compare with the fast tools identified here and stands to a large degree on parametric and random benchmarks. There is also always the possibility that we have prepared the input in a way not ideal for the tool. For instance, transformation to the separated AFA, required by BISIM, is not entirely trivial. Further investigation of this and a comparison with some other implementation of the up-to techniques seems to be needed.[9] The lack of a raw speed of JALTIMPACT on BRE and AFA emptiness is expectable considering that it is meant for a different kind of systems, AFA over data words. The stable trends shown in the graphs suggest that an implementation of an interpolation-based abstraction refinement optimized for BRE and AFA emptiness might have a potential.

Main takeaways. The backward reduction of AFA emptiness to BTS reachability in a combination with IC3 is very fast and extremely versatile, showing very good performance on almost all benchmarks. However, on BRE with a relation to a real world application, simple NFA algorithms actually tend to have the best raw performance, with the simplest implementation of NFA being the best. Antichain algorithms work also well,

[9] It was noted by the author of BISIM that a version of the tool from the year 2016 might be faster. We were however not able to compile it.

even significantly better than other algorithms on specific kinds of AFA. These seem to be the tools to use. Reasonable implementations of the backward BTS reduction with IC3, of antichain, and of basic NFA should also be the baseline of comparisons.

MONA and BRICS, based on DFA, as well as JALTIMPACT focused on data words rather then on pure regular properties, do no reach the performance of the best tools. Also BISIM did not confirm the power of up-to algorithms. SMT-solvers, Z3 especially, are competitive, but cannot be considered the top of state of the art.

Generally, the particular kind and source of benchmark is a decisive factor influencing the performance of tools, as especially visible on the parametric benchmark.

Threads to validity. Our results must be taken with a grain of salt as the experiment contains an inherent room for error. Although we tried to be as fair as possible, not knowing every tool intimately, the conversions between formats and kinds of automata, discussed at the start of Section 5, might have introduced biases into the experiment. Tools are written in different languages and some have parameters which we might have used in sub-optimal way (we use the tools in their default settings), or, in the case of libraries, we could have used a sub-optimal combination of functions. We also did not measure memory peaks, which could be especially interesting e.g. in when the tools are deployed on a cloud. We are, however, confident that our main conclusions are well justified and the experiment gives a good overall picture. The entire experiment is available for anyone to challenge or improve upon [2].

Acknowledgments

This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, and the FIT BUT internal project FIT-S-20-6427.

References

1. The benchmark used in the paper., `https://github.com/VeriFIT/automata-bench`
2. Experiment replication package and additional material, `https://www.fit.vutbr.cz/research/groups/verifit/tools/afa-comparison/`
3. Jaltimpact, `https://github.com/cathiec/JAltImpact`
4. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Proc. of FMCAD'18. IEEE (2018)
5. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In: Proc. of ATVA'19. LNCS, vol. 11781. Springer (2019)
6. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Proc. of TACAS'10. LNCS, vol. 6015. Springer (2010)
7. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Proc. of TACAS'22. Springer (2022)
8. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)

9. Berzish, Murphy: Z3str4: A Solver for Theories over Strings. Ph.D. thesis (2021), `http://hdl.handle.net/10012/17102`
10. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints. In: Proc. of FM'23. Springer (2023)
11. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proc. of POPL'13. ACM (2013)
12. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Proc. of CIAA'08. Springer (2008)
13. Boudet, A., Comon, H.: Diophantine equations, Presburger arithmetic and finite automata. In: Kirchner, H. (ed.) Proc. of CAAP'96. LNCS, vol. 1059. Springer (1996)
14. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal design and safety analysis of AIR6110 wheel brake system. In: Proc. of CAV'15. Springer (2015)
15. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Proc. of FMCAD'07. IEEE (2007)
16. Bradley, A.: IC3 reference implementation: a short, simple, fairly competitive implementation of IC3 (2015), `https://github.com/arbrad/IC3ref`
17. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. of CAV'10. Springer (2010)
18. Brzozowski, J., Leiss, E.: On equations for regular languages, finite automata, and sequential networks. Theoretical Computer Science **10**(1) (1980)
19. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proc. of Symposium on Mathematical Theory of Automata (1962)
20. Büchi, J.R.: Weak Second-Order Arithmetic and Finite Automata, pp. 398–424. Springer New York, New York, NY (1990). `https://doi.org/10.1007/978-1-4613-8928-6_22`, `https://doi.org/10.1007/978-1-4613-8928-6_22`
21. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. of International Congress on Logic, Method, and Philosophy of Science. SUP (1962)
22. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proc. of CAV'14. Springer (2014)
23. Cécé, G.: Foundation for a series of efficient simulation algorithms. In: Proc. of LICS'17. IEEE (2017)
24. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1) (1981)
25. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. of POPL'18 (2018)
26. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. of POPL'19 (2019)
27. Cox, A.: Model Checking Regular Expressions. URL: `https://mosca19.github.io/slides/cox.pdf` (2019), presented at MOSCA'19
28. Cox, A., Leasure, J.: Model checking regular language constraints. CoRR **abs/1708.09073** (2017)
29. D'Anthoni, L.: A symbolic automata library, `https://github.com/lorisdanto/symbolicautomata`
30. D'Antoni, L., Kincaid, Z., Wang, F.: A symbolic decision procedure for symbolic alternating finite automata. Electronic Notes in Theoretical Computer Science **336** (2018)
31. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Proc. of CAV'17. Springer
32. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proc. of POPL'14. ACM (2014)

33. D'Antoni, L., Veanes, M.: Minimization of symbolic tree automata. In: Proc. of LICS'16. ACM (2016)
34. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proc. of IJCAI'13. ACM (2013)
35. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Alaska. In: Proc. of ATVA'08. Springer (2008)
36. Doyen, L., Raskin, J.: Antichain algorithms for finite automata. In: Proc. of TACAS'10. LNCS, Springer (2010)
37. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of SAT'03. LNCS, Springer (2003)
38. Fellah, A., Jürgensen, H., Yu, S.: Constructions for alternating finite automata. International Journal of Computer Mathematics **35** (1990)
39. Fu, C., Deng, Y., Jansen, D.N., Zhang, L.: On equivalence checking of nondeterministic finite automata. In: Proc. of SETTA'17. LNCS, Springer (2017)
40. Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: Unbounded model-checking with interpolation for regular language constraints. In: Proc. of TACAS'13. LNCS, Springer (2013)
41. Ganty, P., Maquet, N., Raskin, J.: Fixed point guided abstraction refinement for alternating automata. Theoretical Computer Science **411**(38-39) (2010)
42. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: Proc. of CAV'16. Springer (2016)
43. Harding, A.: Symbolic strategy synthesis for games with LTL winning conditions. Ph.D. thesis, University of Birmingham (2005)
44. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Proc. of TACAS '95. LNCS, vol. 1019. Springer (1995)
45. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. of FOCS. IEEE (1995)
46. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT'12. LNCS, vol. 7317, pp. 157–171. Springer (2012)
47. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. of POPL'18 **2** (2018)
48. Holík, L., Lengál, O., Síč, J., Veanes, M., Vojnar, T.: Simulation algorithms for symbolic automata. In: Lahiri, S.K., Wang, C. (eds.) Proc. of ATVA'18. Springer (2018)
49. Holík, L., Lengál, O., Šimáček, J., Vojnar, T.: Efficient inclusion checking on explicit and semi-symbolic tree automata. In: Proc. of ATVA'11. LNCS, Springer (2011)
50. Holík, L., Šimáček, J.: Optimizing an LTS-simulation algorithm. Computing and Informatics (7), 1337–1348 (2010)
51. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: PLDI'09. ACM (2009)
52. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford, CA, USA (1971)
53. Hromkovič, J.: On the power of alternation in automata theory. Journal of Computer and System Sciences **31**(1) (1985)
54. Huffman, D.: The synthesis of sequential switching circuits. Journal of the Franklin Institute **257**(3) (1954)
55. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In: Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday. Springer (2004)
56. Iosif, R., Xu, X.: Abstraction refinement for emptiness checking of alternating data automata. In: Proc. of TACAS'18. Springer (2018)

57. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Transactions on Computational Logic **2**(3) (2001)
58. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of ACM **47**(2) (2000)
59. Lengál, O., Šimáček, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: Proc. of TACAS'12. LNCS, vol. 7214. Springer (2012)
60. Li, J., Pu, G., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: SAT-based explicit LTLf satisfiability checking. Artificial Intelligence **289** (2020)
61. Lutterkort, D.: libfa, `https://augeas.net/libfa/`
62. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. of CAV'03. Springer (2003)
63. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)
64. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies. Volume 34. Princeton University Press, Princeton (1956)
65. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. Springer (2008)
66. Muller, D., Saoudi, A., Schupp, P.: Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In: Proc. of LICS. IEEE (1988)
67. Møller, A., et al.: Brics automata library, `https://www.brics.dk/automaton/`
68. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. In: Proc. of CAV'22. Springer (2022)
69. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6) (1987)
70. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Information and Computation **208**, 1–22 (2010)
71. RegExLib.com: The Internet's first Regular Expression Library. `http://regexlib.com/`
72. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Proc. of SPIN'07. Springer (2007)
73. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proc. of PLDI'21. ACM (2021)
74. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Proc. of LPAR'05. Springer (2005)
75. Valmari, A.: Simple bisimilarity minimization in O(m log n) time. Fundamenta Informaticae **105**(3) (2010)
76. Vardi, M.Y.: Nontraditional applications of automata theory. In: Theoretical Aspects of Computer Software. Springer (1994)
77. Vargovčík, P., Holík, L.: Simplifying alternating automata for emptiness testing. In: Proc. of APLAS'21. Springer (2021)
78. Veanes, M.: A .NET automata library, `https://github.com/AutomataDotNet/Automata`
79. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. of ICST'10. IEEE (2010)
80. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: Proc. of CAV'16. LNCS, vol. 9779. Springer (2016)
81. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In: Mycroft, A. (ed.) Proc. of SAS'95. LNCS, vol. 983. Springer (1995)
82. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)