MATA: A Fast and Simple Finite Automata Library

David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síč

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

Abstract. MATA is a well-engineered automata library written in C++ that offers a unique combination of speed and simplicity. It is meant to serve in applications such as string constraint solving and reasoning about regular expressions, and as a reference implementation of automata algorithms. Besides basic algorithms for (non)deterministic automata, it implements a fast simulation reduction and antichain-based language inclusion checking. The simplicity allows a straightforward access to the low-level structures, making it relatively easy to extend and modify. Besides the C++ API, the library also implements a Python binding.
The library comes with a large benchmark of automata problems collected from relevant applications such as string constraint solving, regular model checking, and reasoning about regular expressions. We show that MATA is on this benchmark significantly faster than all libraries from a wide range of automata libraries we collected. Its usefulness in string constraint solving is demonstrated by the string solver Z3-NOODLER, which is based on MATA and outperforms the state of the art in string constraint solving on many standard benchmarks.

1 Introduction

We introduce a new finite automata library MATA[1]. It is intended to be used in applications where automata languages are manipulated by set operations and queries, presumably in a tight loop where automata are iteratively combined together using the classical as well as special-purpose constructions. Examples are applications like *string constraint solving* algorithms such as [11,24,22,1,10,3,71], processing of regular expressions [28,49], *regular model checking* (e.g., [16,15,58,26,13,81,6]), or *decision procedures for logics* such as WS1S or quantified Presburger arithmetic [20,80,43,12]. The solved problems are computationally hard, often beyond the PSPACE-completeness of basic automata problems such as language inclusion. Efficiency is hence a primary concern. Achieving speed in applications requires, on one hand, fast implementation of basic automata algorithms (union, intersection, complement, minimization or size reduction, determinization, emptiness/inclusion/equivalence/membership test, parsing of regular expressions) and, on the other hand, access to low-level primitives to implement diverse application-specific algorithms and optimizations that often build on a tight integration with the application environment. Moreover, processing of regular expressions and, even more so, string constraint solving are areas of active research, with constantly evolving algorithms, heuristics, and optimizations. An automata library hence needs flexibility, extensibility, easy access to the low-level data structures, and

[1] https://github.com/VeriFIT/mata

ideally a low learning curve, which is important when involving students in academic research and utilizing limited resources of small research teams.

Fast and simple are therefore our two main requirements for the library. An additional third requirement is a well-engineered infrastructure and a good set of benchmarks and tests, important for effective research and reliable deployment. Mata is therefore built around a data structure for the transition relation of a non-deterministic automaton that is a compromise between simplicity and speed. It represents transitions explicitly, as triples of a sources state, a single symbol, and a target state. This contrasts with various flavors of symbolic representation of transition relation used in advanced automata implementations in order to handle large or infinite alphabets (e.g. Unicode in processing of texts, or bit vectors in reasoning about LTL, arithmetic, or WS1S). However, in the applications we consider, working internally with large alphabets can essentially always be avoided by preprocessing (mainly by *mintermization*, aka factorization of the alphabet). The simplicity of an explicit representation then seems preferable. It allows to use a data structure specifically tailored for computing post-images of tuples and sets of states in automata algorithms: a source state-indexed array, storing at each index the transitions from that source state in a two layered structure, with the first layer divided and ordered by symbols, and the second layer ordered by target states. The data structure seems to be unique among the existing libraries and yields an exceptional performance.

Mata currently provides basic functionality, basic automata operations and tests, parsing of regexes and automata in a textual format, and mintermization. From the more advanced algorithms for working with non-deterministic automata, it implements antichain-based inclusion checking [35], and simulation-based size reduction based on the advanced algorithm of [65,4,48]. The inclusion check appears to be by a large margin the fastest implementation available, and together with the tree automata library Vata [59], Mata is the only library with an implementation of a simulation algorithm of the second generation originating from [65,21] (the second generation algorithms combine partition-relation pairs to manipulate preorders that were handled explicitly by the first generation algorithms such as [44,52]). Mata is implemented in C++, uses almost exclusively the STL library for its data structures, and has no external dependencies[2] This makes it relatively easy to learn and integrate with other software projects. It is a well-engineered project at GitHub, with modern test and quality of code assurance infrastructure. Besides the C++ API, it provides a Python binding for fast prototyping and easy experimenting, for instance using interactive Jupyter notebooks.

We evaluated its speed in, to our best knowledge, so far the most comprehensive comparison of automata libraries. We compare with 7 well-known automata libraries on a large benchmark of problems from domains close to Mata's designation, mainly string constraint solving, processing regular expressions, and regular model checking. Mata consistently outperforms all other libraries, from several times to orders of magnitude.

That Mata is a good fit for string constraint solving is demonstrated by its central role in the string solver Z3-Noodler, which implements the algorithms of [11,24], and outperforms the state of the art on many standard benchmarks (see [25] for details).

[2] Although, at the moment, it uses the BDD library CUDD [70] in mintermisation and the regular expression parser from RE2 [41]. The code from these projects is, however, contained within Mata. Moreover, the connection to CUDD is not tight and we plan to remove it in the future.

Our contributions can be summarised by the following three points:

1. MATA, a fast, simple, and well-engineered automata library, well suited for application in string constraint solving and regex processing, in research and student projects, as well as in industrial applications.
2. An extension of a benchmark of automata problems from string constraint solving, processing regular expressions, regular model checking, and solving arithmetic constraints.
3. A comparison of a representative sample of well-known automata libraries against the above benchmark, demonstrating the superior performance of MATA.

2 Related Work

In this overview of automata algorithms and implementations, we focus on the technology relevant to MATA, i.e., automata used as a symbolic representation of sets of words and manipulated mainly by set operations. We omit automata technology made for other purposes, such as regular pattern matching, which concentrates on the membership test.

Automata techniques. The most textbook-like approach is to keep finite automata deterministic (the so-called DFA), which has the advantage of simple algorithms and data structures. Essentially all classical problems reduce to product construction, determinization by subset construction, final state reachability test, and minimization (by Hopcroft's [50], Moore's [62], Brzozowski's [19], or Huffman's [51] algorithms). The obvious drawback is the susceptibility to state explosion in determinization.

An alternative is to determinize automata only when necessary (e.g., only before complementing). Non-determinism may bring up to exponential savings in automata sizes and modern algorithms for *nondeterministic finite automata* (NFA) can in practice avoid the exponential worst-case cost of problems like the language inclusion test.

Namely, a major breakthrough in working with NFAs were the antichain-based algorithms for testing language universality and inclusion of NFA first introduced (to the best of our knowledge) in [74] and later rediscovered in [82]. They dramatically improve practical efficiency of the subset construction by subsumption pruning (discarding larger sets). They were later extended with simulation [5,35] (and generalized to numerous other kinds of automata and problems). A principally similar is the bisimulation up-to congruence technique of [14], which optimizes the NFA language equivalence test. Although experimental data in various works are somewhat contradictory, the more systematic studies so far found antichain-based algorithms more efficient [39,38].

NFAs require more involved reduction methods than DFAs, such as those based on simulation [65,21,44,52,48] or bisimulation [76,64,46]. Simulation reduces significantly more but is much more costly. The algorithms for computing simulation of the *second generation* [65,21], which use the so-called partition-relation pairs to represent preorders on states, are practically much faster than the *first generation algorithms* [44,52].

Representations of the transition relation. In order to handle automata over large or infinite alphabets, such as Unicode or bit vectors, some implementations of automata represent transitions symbolically. Transitions may be annotated by sets of symbols represented as BDDs, logical formulae, intervals of numbers, etc. The most systematic approach to this has been taken in works on *symbolic automata* [78,32,33], where the symbol predicates may be taken from any *effective Boolean algebra* (essentially a countable set closed under Boolean operations). Some libraries, such as Spot [36], Owl [57], or Mosel [55] use BDDs to compactly represent sets of symbols on transitions. Even more compact are the symbolic representations of the transition relation used in Mona [43] and in the symbolic version of the tree automata library Vata [59], where all transitions starting at a state are represented as a single multi-terminal BDDs with the target states in the leaves (the paths represent symbols). Although symbolic representation may offer new optimization opportunities [32] and give more generality, it also brings complexity and overhead. Adapting the known algorithms may be nontrivial [32,46] to the point of being a difficult unsolved problem (such as the fast computation of simulation relation of [65,21]). In our application area, working with large alphabets can mostly be avoided in preprocessing, for instance by means of a priori mintermization (partitioning the alphabet into groups of symbols indistinguishable from the viewpoint of the input problem). The simplicity and transparency of explicit representation of transitions then seems preferable.

Alternating automata. Alternating automata (AFA) received attention recently in the context of string solving and regex processing [79,28,45,40]. They allow to keep automata operations implicit up to the point of the PSPACE-complete emptiness test, which can be solved by clever heuristics (e.g. [79,28,45,82,38,30]). Available implementations were recently compared with selected NFA libraries [38] and neither approach dominated. AFA are, however, often not a viable alternative since adapting complex algorithms from, e.g., string solving to AFA typically requires to redesign the entire algorithm from scratch (as, e.g., in [45,79]).

String solving and SMT solvers. String constraint solving is currently the primary application target of Mata. Mata is already a basis of an efficient string solver Z3-Noodler [25] and a number of other string solvers could perhaps benefit from its performance, especially those that already use automata as a primary data structure, e.g. [23,3,10,1]. Besides, SMT string constraint solvers can also be used to reason about regular properties, though the results of [38] suggest that their efficiency is not on par with dedicated fast automata libraries.

Automata libraries. We give overview of known automata libraries with a focus on those that we later include in our experimental comparison in Section 6.

The Brics [63] automata library is often considered a baseline in comparisons. It implements both NFA and DFA, where each state keeps the set (implemented as a hash map) of transitions, which are represented symbolically using character ranges. It is written in Java and relatively optimized.

The Automata.net library [77], written in C#, implements symbolic NFA parameterized by an effective Boolean algebra. The transition relation (as well as its inverse) are

implemented as a hash map from states to the dynamic array of transitions from a given state, each transition annotated with a predicate over the algebra. We use it in our comparison with the algebra of BDDs. AUTOMATA.NET has been developed for a long time and has accumulated a number of novel techniques (e.g., an optimized minimization [31]).

MONA [43], written in C, is a famous optimized implementation of deterministic automata used for deciding WS1S/WSkS formulae. To handle DFA with complex transition relations over large alphabets of bit vectors, MONA uses a compact fully symbolic 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 every operation implicitly determinizes its output.

VATA [59], written in C++, implements non-deterministic tree automata. It can be used with NFA, too as they are a special case of tree automata. It is relatively optimized and features fast implementation of the antichain-based inclusion checking [15,47] (which for NFA boils down to the inclusion check of [35]) and the second generation simulation computation algorithm of [48].

AWALI [60] is a library that targets weighted automata and transducers over an arbitrary semiring. To implement the transition relation, it keeps a vector of transitions and for each state s two vectors: one keeps the indices of transitions leaving s and the other one the indices of transitions entering s.

AUTOMATALIB [53] is a Java automata library and the basis of the automata learning framework LearnLib [54]. It focuses on DFAs and implements their transition relation as a flattened 2D matrix that maps the source state and symbol to the target state.

AUTOMATA.PY [37] is written in Python. It defines the transition relation in a liberal way, as any mapping from source states to a mapping of symbols to a target state (DFA) or to a set of target states (NFA).

FADO [7] is a Python library written with efficiency in mind. It uses a similar structure as AUTOMATA.PY, but more specific, with the transition as a Python dictionary (a hash map), and states represented as numbers used as indices into an array.

There is a number of other automata libraries that we do not include into our comparison since they seem similar to the included ones or we were not able to use them. The C alternative of BRICS [61] and the Java implementation of symbolic NFA of [29] are in our experiment covered by AUTOMATA.NET and BRICS. ALASKA [34] contains interesting implementations of antichain-based algorithms, but is no longer maintained nor available. LASH [12] is a long-developed tool for arithmetic reasoning based on automata, with an efficient core automata library, written in C. Its transition relation is an array indexed by states, where every state is associated with a symbol-target ordered list of transitions. LASH uses partial symbolic representation – it encodes symbols as sequence of binary digits. The comparison with MONA in [56] on automata benchmark originating from arithmetic problems placed its performance significantly behind MONA. It seems to no longer be maintained, and we were not able to run it on our benchmarks.

There is also a number of implementations of automata over infinite words, for instance SPOT [36], OWL [57], or GOAL [75], which are in their nature close to the finite word automata libraries (SPOT and OWL are optimized and use BDDs on transition edges similarly as AUTOMATA.NET), but implement different algorithms.

Mata evolved from a prototype implementation eNfa used in the comparison of AFA emptiness checkers as a baseline implementation of classical automata [38]. Surprised by its performance, we decided to turn it into a serious widely usable library. Current Mata is much more mature and efficient than the eNfa of [38].

3 Preliminaries on Finite Automata

Words and alphabets. An *alphabet* is a set Σ of *symbols/letters* (usually denoted $a, b, c, \ldots$) and the set of all words over Σ is denoted as Σ^*. The *concatenation* of words u and v is denoted by $u \cdot v$. The *empty word*, the neutral element of concatenation, is denoted by ϵ ($\epsilon \notin \Sigma$).

Finite automata. A *(nondeterministic) finite automaton (NFA)* over an alphabet Σ is a tuple $\mathcal{A} = (Q, post, I, F)$ where Q is a finite set of *states*, $post\colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is a *symbol-post function*, $I \subseteq Q$ 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 $p_0 a_1 p_1 a_2 \ldots a_n p_n$ where for all $1 \leq i \leq n$ it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $p_i \in post(p_{i-1}, a_i)$, and $w = a_1 \cdot a_2 \cdots a_n$. 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. $\mathcal{A}$ is called *deterministic (DFA)* if $|I| \leq 1$, $|post(q, \epsilon)| = 0$, and $|post(q, a)| \leq 1$ for each $q \in Q$ and $a \in \Sigma$. A state is *useful* if it belongs to some accepting run, else it is *useless*. An automaton with no useless states is *trimmed*. A state is *reachable* if it appears on a run starting at an initial state. In Mata, we further use $post(q) = \{(a, post(q, a)) \mid post(q, a) \neq \emptyset\}$ to denote the *state-post* of q. We call symbol-post and state-post the *post-image functions*. We also use $q\text{-}a\!\rightarrow\! p$ where $p \in post(q, a)$ to denote *transitions*. The set of all transitions of $\mathcal{A}$ is called the *transition relation* of $\mathcal{A}$ and we denote it by Δ.

Automata operations. In this paragraph we assume automata without ϵ transitions. The *subset construction* generates from $\mathcal{A}$ the DFA $(Q^{\subseteq}, post^{\subseteq}, I^{\subseteq}, F^{\subseteq})$ where $Q^{\subseteq} = \mathcal{P}(Q)$, $I^{\subseteq} = \{I\}$, $F^{\subseteq} = \{S \in Q^{\subseteq} \mid S \cap F \neq \emptyset\}$, and where $post^{\subseteq}(S, a) = \bigcup_{s \in S} post(s, a)$. The automaton for *complement* is obtained from it by complementing $F^{\subseteq}$, i.e., the set of final states is given as $Q^{\subseteq} \setminus F^{\subseteq}$. The *intersection* of two automata $\mathcal{A}_1 = (Q_1, post_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, post_2, I_2, F_2)$ is implemented by their product $(Q_1 \times Q_2, post^{\times}, I_1 \times I_2, F_1 \times F_2)$ where $post^{\times}((q, r), a) = post_1(q, a) \times post_2(r, a)$. A sensible implementation of course only computes the reachable parts of the product and the subset construction. The *union* $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ is obtained by disjointly uniting all components of $\mathcal{A}_1$ and $\mathcal{A}_2$. Similarly, the *concatenation* $L(\mathcal{A}_1).L(\mathcal{A}_2)$ is the automaton $(Q_1 \uplus Q_2, post_1 \uplus post_2 \uplus post', I_1, F')$ where $\uplus$ denotes the disjoint union, $post'(q, a) = \{r \mid q \in F_1 \wedge \exists s \in I_2\colon r \in post_2(s, a)\}$ is the connecting symbol-post and F' is F_2 if $I_2 \cap F_2 = \emptyset$ and $F_1 \cup F_2$ otherwise (this construction avoids introducing ϵ-transitions). Note that we omit superscript of symbol-post function when it is clear from the context.

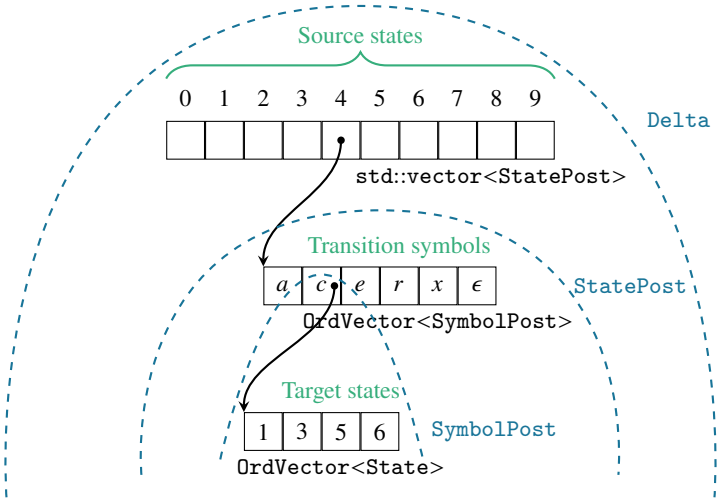


Fig. 1: The transition relation.

4 The Architecture of MATA

We explain in this section the implementation techniques that make MATA efficient on a wide range of automata operations.

4.1 Automata Representation

States and transition symbols are unsigned integers (starting from 0). This makes it easy to store information about them in a state-/symbol-indexed vectors. A frequently used low-level data structure is `OrdVector`, a set of ordered elements implemented as an ordered array (with `std::vector` as the underlying data structure). It has constant time addition and removal of the largest element (`push_back` and `pop_back`), linear union, intersection, and difference (by variants of merging), good memory locality and fast iteration through elements, logarithmic lookup (by binary search), but a slow insertion and removal (`insert` and `erase`) at other than the last position, as the elements on the right of the modified position must be shifted. Many MATA algorithms utilize the constant time handling of the largest element in, e.g., synchronized traversal of multiple `OrdVector` containers. Initial and final states are kept in sparse sets [18], with fast iteration through elements and constant lookup, insertion, and removal.

Data structure for the transition relation. The main determinant of MATA is its three-layered data structure `Delta` for the transition relation. It is implemented as a vector `post` where, for every state q, `post`$[q]$ is of the type `StatePost`, representing $post(q)$ as an `OrdVector` of objects of the type `SymbolPost`, each in turn representing one $post(q, a)$

by storing the symbol a and an `OrdVector` of the target states. The `SymbolPosts` in `OrdVector` are ordered by their symbols[3]. A visualization of `Delta` is shown in Fig. 1.

The weak point of `Delta` is inherited from `OrdVector`: slow `insert` or `erase` of a specific transition (these operations are, however, used scarcely in the considered scenarios). Its strength is mainly fast iteration through the post-image of a state, of a pair of states in the product construction, and of a set of states in the subset construction.

4.2 Automata Operations

Generating post-images in subset construction. In the subset construction, each iteration through $post(S)$ for a set of states S is keeping an array of iterators, one into each $post(q)$ for all $q \in S$. Every iteration shifts the iterators to the right, to $post(q, b)$ where b is the closest from above to the current global minimal symbol a, and returns $post(S, a)$ as the union of all $post(q, a)$'s pointed to by the iterators. No searching in vectors is needed. The entire iteration through all $post(S)$'s makes the iterators in the `SymbolPosts` traverse their respective vectors only once.

Constructing the transitions leading from S while iterating through $post(S)$ is done by appending to `OrdVectors`, without a need to insert at internal positions of vectors. The iteration through the `SymbolPosts` is ordered by symbol, hence each newly created transition from the macrostate S has a larger symbol than all the previously created ones. The symbol-post therefore belongs at the end of the `OrdVector` of symbol-posts of $post(S)$, where it is `push_back`ed. Since the resulting automaton is deterministic, the vectors of targets are singletons, and their creation does not require `insert` either.

Generating post-images in product construction. Similarly as in the subset construction above, iterating through $post((q, r))$ in the product construction is done by synchronous iteration through $post(q)$ and $post(r)$ from the smallest common symbol to the largest. In each step, the iteration returns the Cartesian product of the targets in the symbol-posts. Unlike the subset construction, adding the corresponding transitions from (q, r) to the product automaton sometimes does need an `insert` into the vector of targets. It is however not that frequent: Newly discovered product states are assigned the so far highest numbers, so these are added to the target vectors by `push_back`. The `insert` may hence be needed only when creating a non-deterministic transition to a state discovered earlier.

Storing sets and pairs of states in the subset and product construction. `OrdVector` is also used to map generated sets in the subset to the identities of generated states. The map uses a hash table (`std::unordered_map`) where values are `OrdVectors`. The product construction uses either a two-dimensional array to map pairs of states to product states (for smaller automata) or a vector `pro_map` of hash tables, where the identity of the product state (q, r) is found in the hash map `prod_map[q]` under the key r.

[3] MATA supports ϵ-transitions and some operations can work with them internally. We represent ϵ as the symbol with the highest possible number, hence `SymbolPost` with ϵ is always the last one in the vectors of `SymbolPosts` in `Delta`. The ϵ is therefore easy to be accessed in, e.g., ϵ-transition elimination. Some operations also support several ϵ-like symbols (e.g., ϵ_1, ϵ_2, ...), which are convenient in some algorithms in string solving [11,24] or can play a role of different synchronization symbols, etc.

Emptiness test and trimming. Emptiness test and trimming are used frequently and must be fast. MATA's emptiness test is just a state space exploration that utilizes the fast iteration through post-images of a state.

Trimming consists of two steps: (1) identification of useful states and (2) removal of states that are not useful. Identification of useful states must, besides forward exploration to identify reachable states, identify states that reach a final state. A naive solution would be a backward exploration from final states. `Delta` is, however, not well suited for backward search and although reverting it is doable, its cost is not negligible either. We therefore use a smarter solution, which uses a simplification of the non-recursive Tarjan's algorithm [73] to discover strongly connected components (SCCs). Tarjan's algorithm is essentially a depth-first exploration augmented to identify the SCCs. To identify useful states, on finding an SCC with a final state, we mark the entire SCC as useful together with all states on the path to that SCC, which is readily stored on the depth-first search stack. The cost of computing useful states is then similar to the cost of a single depth-first exploration, which is indeed negligible.

Removal of useless states then needs to be done in a `Delta`-friendly way. The naive approach that removes useless states and transitions incident with them one by one would be extremely slow due to the need of searching and calling `erase` in the `OrdVector`s of `Delta`. Instead, we perform the whole removal and related operations in a single pass through `Delta`. Before the pass begins, first, we create a map `renaming` mapping each useful state to its new name (the trimming also renames the states in order to have the remaining states form a consecutive sequence). During the pass, the following operations need to be performed: (i) in the outermost loop, each useful state q in `Delta` is moved to index `renaming`$[q]$, (ii) in every vector of target states, each useful target is moved to the left in the target vector by that many positions, as there were smaller useless states before it, and (iii) while doing that, the target state q is renamed to `renaming`$[q]$.

Union and concatenation. MATA is relatively slow in operations that copy or create large parts of automata, such as non-deterministic union or concatenation, or simple copying of an automaton. This is perhaps due to the imperfect memory locality (the three layers of vectors in `Delta`) and the need to copy every single transition (unlike, e.g., symbolic automata with BDDs on transitions, where the BDDs may be shared). MATA has, however, in-place variants of union and concatenation, which do not copy `Delta`, but only append the `post` vectors and rename the target states in the appended part, which is fast. The price for the speed is the loss of the original automata, but they are in many use cases not needed (as, e.g., in inductive constructions of automata from regular expressions or formulae).

Antichain-based inclusion checking. MATA implements the antichain-based inclusion checking of [35]. Given the inclusion problem $L(\mathcal{A}) \subseteq L(\mathcal{B})$, the algorithm explores the space of the product of $\mathcal{A}$ and the subset construction on $\mathcal{B}$, consisting of pairs (q, S) with q being a state of $\mathcal{A}$ and S being a set of states of $\mathcal{B}$. In particular, it searches, on the fly, for a reachable pair (q, S) with a final q and a non-final S, which would be a witness non-inclusion. The algorithm optimizes the search by *subsumption pruning*—discarding states (q, S) if another (q, S') with $S \subseteq S'$ has been found. Our implementation uses the infrastructure for computing post-images of product and subset construction discussed

above. The reached pairs (q, S) are stored in a state-q-indexed vector `incl_map` of collections of sets S. The sets are again represented as `OrdVectors`. On reaching a pair (q, S), all sets S' stored in `incl_map`$[q]$ are tested for inclusion with S. If $S \supseteq S'$, then S is dropped, and if $S \subseteq S'$, then S' is removed from `incl_map`$[q]$ (as well as other sets S'' such that $S \subseteq S''$) and S is added to `incl_map`$[q]$. A large speed-up is sometimes obtained by prioritizing exploration of pairs (q, S) with S being of a small size. A smaller set means a better chance to subsume other pairs, to reach a witness of non-inclusion, and to generate other pairs with small sets. The algorithm then explores a much smaller state space.

Simulation. Mata uses an implementation of a fast algorithms for computing simulation, namely, the algorithm from [65], which was adapted from Kripke structures to automata in [4], and later further optimized in [48]. The implementation originates in Vata [59].

Low-level API. The API of Mata contains an interface for accessing the most low-level features needed to implement algorithms in the style described above. For instance, the API provides iterators over transitions of Δ in the form of triples $q\text{-}a\!\rightarrow\! r$, iterators through *moves* (pairs (a, r) such that $q\text{-}a\!\rightarrow\! r \in \Delta$) of a state q, or generic *synchronized iterators*, which allow a simultaneous iteration in a set of vectors used in union and in computing the post-image in the product and subset construction. Since the main data structures are not complicated and have simple invariants, programming with them on the low level is possible even for an outsider. This low-level Mata API is, for instance, used in the string solver Z3-Noodler. [25] presents a detailed comparison of Z3-Noodler with the state of the art in string solving. Its exceptional performance on regex and word equation-heavy constraints is to a large degree due to Mata.

5 Infrastructure of Mata

Mata comes with the following tools and features to make using, developing, and extending it convenient.

Python interface. Mata provides an easy-to-use Python interface, making it a full-fledged automata library for Python projects. It is available on the official Python package repository[4] and can be installed easily using the `pip` package manager:

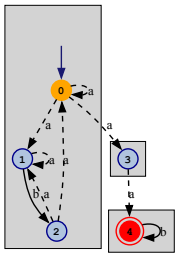
```
$ pip install libmata
```

An example of using the Mata Python binding is shown in Fig. 2. The interface is implemented using the optimizing static compiler Cython wrapping the C++ Mata calls and covers all important parts of the C++ functionality. This low-level interaction with the optimized C++ code keeps the Python code fast. To show the capabilities of the interface and to provide material for easy onboarding, Mata also contains several Jupyter notebooks with examples of how to use it.

[4] `https://pypi.org/project/libmata/`

```
from libmata import nfa, alphabets, parser, plotting
aut1 = parser.from_regex('((a+b)*a)*')
aut2 = parser.from_regex('aab*')
con_aut = nfa.nfa.concatenate(aut1, aut2).trim()
plotting.store()['alphabet'] = \
    alphabets.OnTheFlyAlphabet.from_symbol_map({'a':97, 'b':98})
e_h = [
    (lambda aut, e: e.symbol == 98, {'color':'black'}),
    (lambda aut, e: e.symbol == 97, {'style':'dashed','color':'black'})
]
n_h = [
    (lambda aut, q: q in aut.final_states,
        {'color':'red','fillcolor':'red'}),
    (lambda aut, q: q in aut.initial_states,
        {'color': 'orange', 'fillcolor': 'orange'}),
]
plotting.plot(con_aut, with_scc=True,
              node_highlight=n_h, edge_highlight=e_h)
```

(a) An example of using MATA from Python.



(b) The output.

Fig. 2: An example of a Python interface for MATA. The code (a) loads automata from regular expressions (`a`, `b` are transition symbols; $\star$, and + represent iterations: 0 or more, and 1 or more, respectively), concatenates them, and displays the trimmed concatenation using the conditional formatting with the output in (b).

`.mata` *format and parsing.* MATA brings its own automata format. The main features of the format are extensibility to cover various types of automata, human-readability, yet still high level of compactness. Each `.mata` file consists of automata definitions. The first line of the definition describes the type of the automaton, together with the alphabet. The format supports both explicit and symbolic (bit vector) alphabets. For a symbolic alphabet, symbols are encoded as formulae over atomic propositions, where the parser of `.mata` implements *mintermization* (partitioning the alphabet into groups of symbols indistinguishable from the viewpoint of the input problem), which transforms it into an explicit alphabet with the symbols representing the minterms. The following lines contain a sequence of key-values statements that set particular traits of the automaton, such as initial or final states. The rest of the definition is a list of transitions. Examples of automata in `.mata` format are shown in Fig. 3.

```
@NFA-explicit
%Initial q0 q1
%Final q1
q0 a48 q1
q0 a52 q1
q1 a48 q1
```

(a) NFA with explicit alphabet.

```
@NFA-bits
%Initial q1
%Final q2 q1 q0
q0 ((!a0 | !a1) & a2) q2
q1 (a0 & a1 & !a2) q0
q2 ((a0 & a1) | a2) q1
```

(b) NFA with symbolic alphabet.

Fig. 3: Examples of NFAs in the `.mata` format.

Other than the introduced format, MATA can also parse automata from regular expressions using the parser from the regex matcher RE2 [41]. This means that MATA can handle even complex syntax used in real-world regular expressions.

Continuous integration. We implement continuous integration via GitHub Actions. In particular, actions automatically build the library including the Python binding on MacOS and Ubuntu, check for warnings, code quality and run unit tests together with the code coverage. The actions are triggered after each commit, and the checks are mandatory for merging branches to the main branch, and can also be run locally.

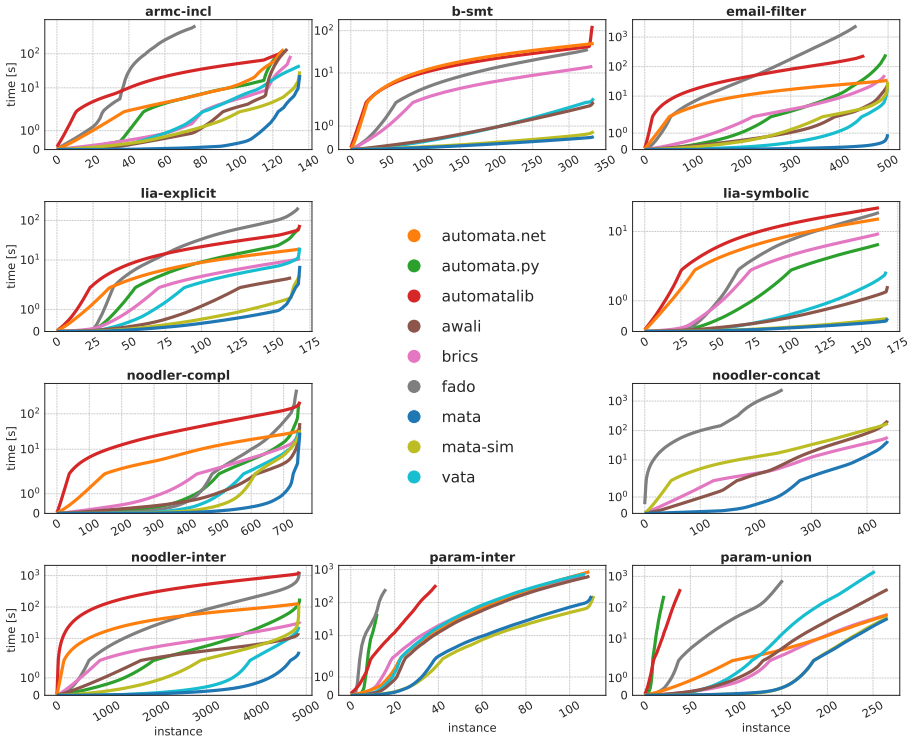


Fig. 4: Cactus plot showing cumulative run time per benchmark. The time axis is logarithmic.

6 Experimental Evaluation

We compared Mata against 7 selected libraries discussed in Section 2: Vata [59], Brics [63], Awali [60], Automata.net [77], AutomataLib [53], FAdo [7], and Automata.py [37],[5] on a benchmark of basic automata problems from string constraint solving, reasoning about regular expressions, regular model checking, and a few examples from solving arithmetic formulae. Most of the benchmark problems are taken from earlier works [38,30,40,28], but we added new problems from string constraint solving and solving quantified linear integer arithmetic (LIA).

We mainly aim to demonstrate the efficiency of the basic data structures and implementation techniques of Mata. This is best seen on standard constructions, where all libraries implement the same high-level algorithm, such as product, subset construction, or reachability test within complementation, intersection, emptiness test, etc. We then also showcase the efficiency of more advanced algorithms implemented only in Mata and Vata, the antichain-based inclusion test and simulation reduction.

[5] We also tried to compare with Mona, but using it as a standalone library that would parse automata in our format turned to be problematic. We were getting many inconsistent results and so we decided to drop it from the comparison.

Table 1: Statistics for the benchmarks. We list the number of timeouts (TO), average time on solved instances (*Avg*), median time over all instances (*Med*), and standard deviation over solved instances (*Std*), with the best values in **bold**. The times are in milliseconds unless seconds are explicitly stated. We use ∼0 to denote a value close to zero.

	armc-incl (136)				**b-smt** (384)				**email-filter** (500)				**lia-explicit** (169)				**lia-symbolic** (169)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
MATA	**0**	**174**	**2**	1 s	**0**	**1**	**1**	1	**0**	**1**	**∼0**	9	**0**	42	**6**	356	**0**	**2**	**2**	6
AWALI	7	1 s	17	3 s	**0**	6	6	4	**0**	46	4	162	6	**21**	21	16	**0**	8	7	14
VATA	**0**	324	43	577	**0**	7	7	10	**0**	42	2	322	**0**	121	51	671	1	11	10	11
AUTOMATA.NET	9	1 s	125	3 s	**0**	148	153	30	**0**	69	66	30	**0**	113	117	49	6	103	107	33
BRICS	5	659	34	2 s	4	43	43	19	6	103	17	280	**0**	66	62	63	6	55	60	33
AUTOMATALIB	10	843	669	1 s	7	390	126	3 s	48	516	390	521	**0**	458	285	1 s	6	164	173	52
FADO	58	8 s	22 s	10 s	9	109	112	67	64	6 s	1 s	11 s	1	1 s	727	2 s	6	135	149	105
AUTOMATA.PY	10	913	133	3 s	334	24	TO	15	4	520	19	2 s	1	372	167	894	6	35	35	25

	noodler-compl (751)				**noodler-conc** (438)				**noodler-inter** (4872)				**param-inter** (267)				**param-union** (267)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
MATA	**0**	**39**	**∼0**	401	**0**	**100**	**10**	286	**0**	**∼0**	**∼0**	3	**156**	**1 s**	TO	4 s	**0**	**166**	**7**	326
AWALI	**0**	73	2	638	**0**	490	55	1 s	6	3	1	7	157	6 s	TO	7 s	**0**	1 s	81	3 s
VATA	**0**	57	2	296		-			2	4	**∼0**	22	159	7 s	TO	8 s	14	6 s	270	12 s
AUTOMATA.NET	**0**	53	39	110		-			**0**	26	24	9	157	8 s	TO	10 s	**0**	220	47	314
BRICS	**0**	47	8	190	**0**	136	35	204	**0**	7	3	21	159	6 s	TO	6 s	**0**	223	50	307
AUTOMATALIB	**0**	293	143	793		-			17	276	216	675	227	8 s	TO	13 s	227	10 s	TO	15 s
FADO	10	646	5	4 s	189	10 s	25 s	13 s	10	271	52	2 s	250	15 s	TO	20 s	115	5 s	12 s	11 s
AUTOMATA.PY	3	263	5	2 s		-			5	38	3	353	254	4 s	TO	6 s	245	11 s	TO	16 s

Benchmarks. We use the following benchmark sets.

b-smt [38] contains 384 instances of boolean combinations of regular properties, obtained from SMT formulae over the theory of strings. These include difficult hand-written problems containing membership in regular expressions extended with intersection and complement from [71] and emptiness problems from Norn [2,3] and SyGuS-qgen benchmarks, collected in SMT-LIB [9,67,68].

email-filter [38] contains 500 inclusion checks of the form $r_5 \subseteq r_1 \wedge r_2 \wedge r_3 \wedge r_4$ obtained analogously as in [30]. Each r_i is one of the 75 regexes[6] from RegExLib [66], selected so that $r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$ is not empty. Similar kind of these problems is solved in spam-filtering: one tests whether a new filter r_5 adds anything new to existing filters.

param-inter [38] contains 4 sets of parametric intersection problems from [40] and 2 sets from [28]. In total, this includes 267 problems. The parameter controls the size of the regex or the number of regexes to be combined. **param-union** is the variant of the benchmark that performs union instead of intersection.

armc-incl [38] contains 136 language inclusion problems derived from runs of an abstract regular model checker of [15] (verification of the bakery algorithm, bubble sort, and a producer-consumer system).

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

Table 2: Relative speedup of MATA on instances where both libraries finished.

	AWALI	VATA	AUTOMATA.NET	BRICS	AUTOMATALIB	FADO	AUTOMATA.PY
armc-incl	27.52	1.86	29.73	16.98	21.44	4839.55	23.22
b-smt	3.7	4.52	89.64	26.13	236.36	70.16	24.47
email-filter	25.07	22.59	37.19	55.3	273.35	9999.29	282.41
lia-explicit	2.22	2.88	2.69	1.57	10.89	85.17	25.38
lia-symbolic	3.46	4.65	51.82	27.99	82.47	67.54	17.97
noodler-compl	1.85	1.45	1.37	1.22	7.44	137.53	15.58
noodler-conc	4.87	-	-	1.36	-	1979.56	-
noodler-inter	4.02	6.42	33.98	9.04	371.23	363.49	51.51
param-inter	5.36	7.3	7.27	6.49	1.43	2148.64	58.85
param-union	8.61	51.77	1.33	1.34	833.69	1618.04	5860.62

lia consists of 169 complementation problems created during the run of Amaya [8], a tool for deciding linear integer arithmetic (LIA) formulae using an automata-based decision procedure of [17]. The formulae are taken from *UltimateAutomizer* [42] and *tptp* [72] benchmarks, collected in SMT-LIB [9,69]. The transition relation in Amaya is represented symbolically using BDDs; in our experiments we tested both symbolic representation (in **lia-symbolic**) and explicit representation (in **lia-explicit**), where explicit symbols are bit vectors represented by the BDDs.

noodler consists of instances created during the run of the string solver Z3-NOODLER [11,24,25] on the regex-heavy benchmark AutomatArk [10] from SMT-LIB [9,67]. We collected 751 complementation, 438 concatenation, and 4,872 intersection problems in **noodler-compl**, **noodler-conc**, and **noodler-inter** respectively.

Experimental setup. We converted all benchmarks into a common textual automata format (the `.mata` format, see Section 5), and wrote dedicated parsers or conversions for all the libraries. The conversion and parsing are not included in the run times since the parsers are not optimized and the typical use cases do not require parsing every input automaton from a textual format. From some of the benchmarks, we excluded small units of examples where the conversion failed. We measure only the time needed for carrying out the specified operations on automata already parsed into each library's internal data structures. Automata in all benchmarks but **lia** and those coming from regexes, **email-filter**, **b-smt**, **param-inter**, and **param-union**, had small or moderate alphabet sizes (all below 100 symbols, except **noodler-inter** with up to 252 symbols). The explicit automata from LIA solving (**lia-explicit**) have at most 1,024 symbols (corresponding to 10 bits).[7] After performing mintermization on automata with symbolic representation (**lia-symbolic**), the number of symbols was reduced to at most 30, and mintermization runs on automata from regular expressions returned alphabets with at most 80 symbols.

[7] It should be noted that these LIA problems are by no means representative of typical LIA formulae, which could generate much larger alphabets and transition relations that require some sort of symbolic representation.

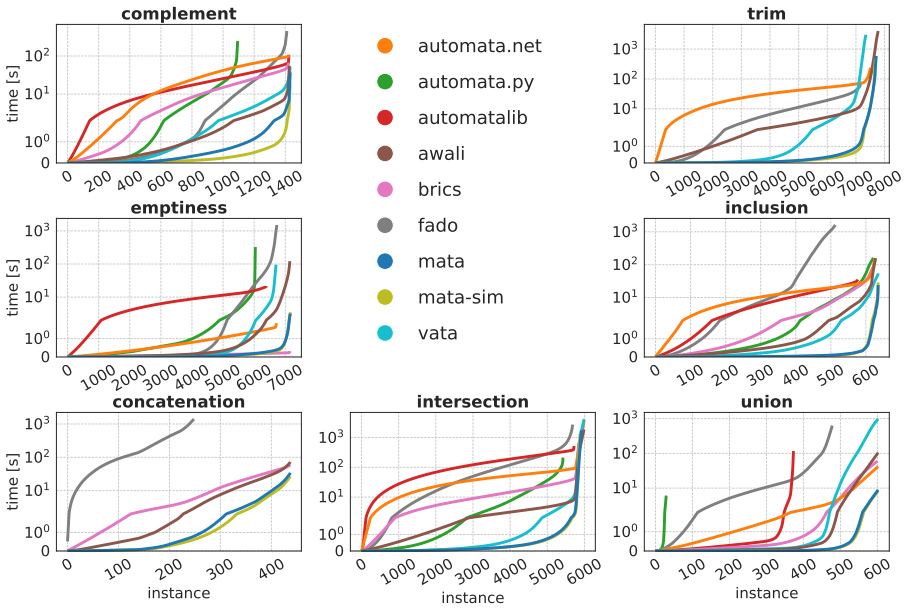


Fig. 5: Cactus plot showing cumulative run time per operation. The time axis is logarithmic.

Results. We summarize the results of each benchmark in cactus plots in Fig. 4 (displaying cumulative run times of benchmarks, with the instances ordered by their run time) and Table 1. Table 2 shows relative speedups of MATA over each library on problem instances that both libraries finished in time. We also present statistics for individual automata operations across the entire benchmark in Fig. 5 and Table 3. We do not show the performance of MATA's Python interface in the plots and tables as it is matches that one of MATA. All examples were run in six parallel jobs on Fedora GNU/Linux 38 with an Intel Core 3.4 GHz processor and 20 GiB RAM with 60 s timeout.

MATA consistently outperforms all other libraries on all benchmarks and in all operations, up to few exceptions. It is sometimes matched or outperformed by AUTOMATA.NET and BRICS in union and concatenation operation (on **param-union** and **noodler-conc**). BRICS and AUTOMATA.NET are sometimes faster since they may be able to share parts of the representation (such as BDDs on the transitions) between the automata operands and the union/concatenation, while MATA copies the entire data structure (and the memory locality of `Delta`, with its three layers of vectors, is not perfect). BRICS appears particularly fast in emptiness checking since it implicitly trims the automata, after which the emptiness test becomes a trivial query on emptiness of the set of states. The cost of the emptiness check is thus hidden in the cost of other operations (we do not state statistics from trimming for BRICS for this reason). BRICS and AUTOMATA.NET also have a smaller average time in constructing the complements in **lia-symbolic**, due to a few high run times of MATA on examples that have many transitions per a pair of states. Solving these examples, and generally examples generated from solving LIA, is indeed

Table 3: Statistics for the operations on solved instances. We list the average time (*Avg*), median time (*Med*), and standard deviation (*Std*), with the best values in **bold**. The times are in milliseconds. Note that only the operations that the given library finished within the timeout are counted, hence the numbers are significantly biased in favour of libraries that timeouted more (the harder benchmarks are no counted in), and should be red in the context of Table 1 and the cactus plots. We use ~0 to denote a value close to zero.

	complement			**concatenation**			**emptiness**			**inclusion**			**intersection**			**trim**			**union**		
	Avg	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*
MATA	**25**	**1**	315	**78**	**8**	235	**~0**	**~0**	2	**37**	**~0**	576	295	**~0**	3 s	76	**~0**	828	**14**	**~0**	45
AWALI	38	2	462	166	22	402	17	**~0**	138	250	2	2 s	312	**~0**	2 s	516	**~0**	4 s	173	**~0**	527
VATA	36	3	294		-		14	**~0**	130	85	1	374	699	**~0**	4 s	408	**~0**	3 s	2 s	**~0**	5 s
AUTOMATA.NET	73	59	89		-		**~0**	**~0**	~0	245	43	1 s	621	14	4 s	31	9	165	69	6	163
BRICS	46	24	140	136	35	204	**~0**	**~0**	~0	204	10	1 s	115	4	1 s		-		99	2	232
AUTOMATALIB	75	31	657		-		3	2	5	60	42	102	91	59	748		-		311	2	3 s
FADO	320	3	2 s	6 s	10 s	10 s	223	**~0**	2 s	3 s	84	8 s	479	48	3 s	**10**	3	70	1 s	84	6 s
AUTOMATA.PY	226	25	2 s		-		53	**~0**	1 s	263	6	1 s	**39**	2	479		-		203	TO	377

a case for symbolic representation of transitions, and it is currently not a primary target of MATA. However, MATA is still much faster than any other library on mintermised versions of the same examples. AUTOMATALIB is faster in some parametric intersection examples because of its implicit determinization, which in some particular examples returns much smaller automata. When the other libraries are made to determinize, they behave analogously, and MATA again solves most examples and takes the least time. Still, on all operations except emptiness, MATA is the fastest overall, and on emptiness it is by far the fastest from libraries that actually do solve the emptiness problem. MATA has especially efficient inclusion test, and trimming, an operation which is usually needed very frequently, is also a strong point of MATA's performance.

MATA's simulation reduction (MATA-SIM in the results) does not help much when the time for computing the simulation is counted in, as seen in Fig. 4. Simulation reduction is indeed costly, and our eager strategy of reducing all automata is probably sub-optimal. The run times of complement, however, show a considerable speedup after automata are reduced, and MATA-SIM solves some complement and also parametric intersection examples that no other library can.

Overall, MATA appears significantly faster than all the libraries we have tried, with the closest competitor being often more than an order of magnitude slower.

Threats to validity. Our results must be taken with a grain of salt as the experiment contains an inherent room for error. Mainly, not knowing every library intimately, we might have missed the most optimal solutions, and our parsers of the `.mata` format might be building the internal data structures of the libraries in a sub-optimal way. The experiment was also running in parallel on a server with limited resources, which might lead to fluctuations in run times We are, however, confident that our main conclusions are well justified.

7 Conclusions and Future Work

We have introduced a new automata library MATA, explained its principles, and evaluated its performance. MATA is not the most general or feature-full library. Libraries such as AWALI or AUTOMATA.NET are much more complex and comprehensive, are more widely applicable, either to various symbolic representations of automata or to automata with registers, while still being impressively efficient. MATA, however, does what it is meant to do better than all the other libraries: solve examples from string solving, regular expression processing, and regular model checking much faster, while staying simple and transparent, easily extensible and applicable to projects.

We continue working on MATA's set of features as well as its efficiency. We plan to extend MATA with transducers, add support for registers that could handle, e.g., counting in regular expressions, and experiment with the poor man's symbolic representation of bit vector alphabets represented as sequences of bits (used in LASH [12]), so that MATA can be used adequately in applications such as solving WS1S and arithmetic formulae. We believe that the efficiency of the basic data structures discussed here can be much improved by focusing on the low-level performance. Custom data structures, specialised memory management, improvement in memory locality, and, generally, the class of optimizations used in BDD packages, could shift MATA's performance much further.

Acknowledgments

This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151.

Data Availability Statement

An environment with the tools and data used for the experimental evaluation in the current study is available at [27].

References

1. 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)
2. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10, https://doi.org/10.1007/978-3-319-08867-9_10
3. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Computer Aided Verification. pp. 462–469. Springer International Publishing, Cham (2015)

4. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing simulations over tree automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 93–108. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_8, https://doi.org/10.1007/978-3-540-78800-3_8
5. 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)
6. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory. pp. 35–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
7. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: Fado and guitar: Tools for automata manipulation and visualization. In: Maneth, S. (ed.) Implementation and Application of Automata. pp. 65–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
8. authors, A.: Amaya (2023), https://github.com/MichalHe/amaya
9. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
10. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 289–312. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_14, https://doi.org/10.1007/978-3-030-81688-9_14
11. 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)
12. Boigelot, B., Latour, L.: Counting the solutions of Presburger equations without enumerating them. Theoretical Computer Science **313**(1), 17–29 (2004). https://doi.org/https://doi.org/10.1016/j.tcs.2003.10.002, https://www.sciencedirect.com/science/article/pii/S0304397503005322, implementation and Application of Automata
13. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification. pp. 223–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
14. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proc. of POPL'13. ACM (2013)
15. 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)
16. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3114, pp. 372–386. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_29, https://doi.org/10.1007/978-3-540-27813-9_29
17. Boudet, A., Comon, H.: Diophantine equations, Presburger arithmetic and finite automata. In: Kirchner, H. (ed.) Trees in Algebra and Programming — CAAP '96. pp. 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
18. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Lett. Program. Lang. Syst. **2**(1–4), 59–69 (mar 1993). https://doi.org/10.1145/176454.176484, https://doi.org/10.1145/176454.176484

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. Cécé, G.: Foundation for a series of efficient simulation algorithms. In: Proc. of LICS'17. IEEE (2017)
22. 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)
23. 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)
24. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Solving string constraints with lengths by stabilization. Proc. ACM Program. Lang. **7**(OOPSLA2) (oct 2023). https://doi.org/10.1145/3622872
25. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Proc. of TACAS'24. LNCS, Springer (2024)
26. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 76–83. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102244, https://doi.org/10.23919/FMCAD.2017.8102244
27. Chocholatý, D., Fiedor, T., Havlena, V., Holík, L., Hruška, M., Lengál, O., Síč, J.: A replication package for reproducing the results of paper "MATA: A fast and simple finite automata library" (Oct 2023). https://doi.org/10.5281/zenodo.10044515, https://doi.org/10.5281/zenodo.10044515
28. Cox, A., Leasure, J.: Model checking regular language constraints. CoRR **abs/1708.09073** (2017)
29. D'Antoni, 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.: Minimization of symbolic automata. In: Proc. of POPL'14. ACM (2014)
32. D'Antoni, L., Veanes, M.: Minimization of symbolic tree automata. In: Proc. of LICS'16. ACM (2016)
33. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 47–67. Springer International Publishing, Cham (2017)
34. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Alaska. In: Proc. of ATVA'08. Springer (2008)
35. Doyen, L., Raskin, J.: Antichain algorithms for finite automata. In: Proc. of TACAS'10. LNCS, Springer (2010)
36. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Gbaguidi Aisse, A., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 174–187. Springer International Publishing, Cham (2022)
37. Evans, C.: Automata (2023), https://github.com/caleb531/automata
38. Fiedor, T., Holík, L., Hruska, M., Rogalewicz, A., Síc, J., Vargovčík, P.: Reasoning about regular properties: A comparative study. In: Pientka, B., Tinelli, C. (eds.) Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy,

July 1-4, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14132, pp. 286–306. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_17, https://doi.org/10.1007/978-3-031-38499-8_17
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. Google: Re2. https://github.com/google/re2
42. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 36–52. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
43. 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)
44. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. of FOCS. IEEE (1995)
45. 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)
46. 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)
47. 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)
48. Holík, L., Šimáček, J.: Optimizing an LTS-simulation algorithm. Computing and Informatics **29**(6+), 1337–1348 (2010), https://arxiv.org/abs/2307.04235
49. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: PLDI'09. ACM (2009)
50. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford University, Stanford, CA, USA (1971)
51. Huffman, D.: The synthesis of sequential switching circuits. Journal of the Franklin Institute **257**(3) (1954)
52. 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)
53. Isberner, M., Howar, F., Steffen, B.: AutomataLib, https://learnlib.de/projects/automatalib/
54. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 487–495. Springer International Publishing, Cham (2015)
55. Kelb, P., Margaria, T., Mendler, M., Gsottberger, C.: MOSEL: A sound and efficient tool for M2L(Str). In: Grumberg, O. (ed.) Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1254, pp. 448–451. Springer (1997). https://doi.org/10.1007/3-540-63166-6_45, https://doi.org/10.1007/3-540-63166-6_45
56. Klaedtke, F.C.: Automata-based decision procedures for weak arithmetics. Ph.D. thesis, University of Freiburg, Freiburg im Breisgau, Germany (2004), http://freidok.ub.uni-freiburg.de/volltexte/1439/index.html
57. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis. pp. 543–550. Springer International Publishing, Cham (2018)

58. Legay, A.: T(O)RMC: A tool for (ω)-regular model checking. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 548–551. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
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. Lombardy, S., Marsault, V., Sakarovitch, J.: Awali, a library for weighted automata and transducers (version 2.0) (2021), software available at http://vaucanson-project.org/Awali/2.0/
61. Lutterkort, D.: libfa, `https://augeas.net/libfa/`
62. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies. Volume 34. Princeton University Press, Princeton (1956)
63. Møller, A., et al.: Brics automata library, `https://www.brics.dk/automaton/`
64. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6) (1987)
65. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Information and Computation **208**, 1–22 (2010)
66. RegExLib.com: The Internet's first Regular Expression Library. `http://regexlib.com/`
67. SMT-LIB: `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S` (2023)
68. SMT-LIB: `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA` (2023)
69. SMT-LIB: `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LIA` (2023)
70. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0 (2015)
71. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proc. of PLDI'21. ACM (2021)
72. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
73. Tarjan, R.E.: Depth-first search and linear graph algorithms (working paper). In: 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971. pp. 114–121. IEEE Computer Society (1971). `https://doi.org/10.1109/SWAT.1971.10`, `https://doi.org/10.1109/SWAT.1971.10`
74. Tozawa, A., Hagiya, M.: XML schema containment checking based on semi-implicit techniques. In: Ibarra, O.H., Dang, Z. (eds.) Implementation and Application of Automata, 8th International Conference, CIAA 2003, Santa Barbara, California, USA, July 16-18, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2759, pp. 213–225. Springer (2003). `https://doi.org/10.1007/3-540-45089-0_20`, `https://doi.org/10.1007/3-540-45089-0_20`
75. Tsay, Y.K., Chen, Y.F., Tsai, M.H., Wu, K.N., Chan, W.C.: Goal: A graphical tool for manipulating büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 466–471. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
76. Valmari, A.: Simple bisimilarity minimization in O(m log n) time. Fundamenta Informaticae **105**(3) (2010)
77. Veanes, M.: A .NET automata library, `https://github.com/AutomataDotNet/Automata`
78. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. of ICST'10. IEEE (2010)
79. 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)

80. 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)
81. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification. pp. 88–97. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
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)