BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Awesome Automata: Algorithms and Applications

Habilitation Thesis

Ondřej Lengál

Brno, Summer 2024

Abstract

In this habilitation thesis, the author focuses on methods for handling various finite-state automata models and their use in applications. Finite-state automata are a basic computational model. Their simplicity and nice theoretical properties make them popular with computer scientists. Although studied for over 80 years, some of their basic questions—e.g., how to complement them or test language inclusion efficiently—remain open. Successful uses of these automata in practical applications usually need significant changes of standard (textbook) algorithms to scale and be efficient. This thesis gives an overview of the author's results in the area of development of efficient algorithms for handling finite-state automata over (in-)finite words and trees and the use of these automata in applications like pattern matching in network intrusion detection systems, deciding logics (the monadic second order theory over words/trees, the first order theory of the addition over integers, and the theory of strings), as well as analysis and simulation of quantum circuits.

Keywords

Finite automata, finite tree automata, Büchi automata, ω-automata, complementation, termination checking, network intrusion detection system, theory of strings, regular model checking, monadic second-order logic, WS1S, WS*k*S, Presburger arithmetic, linear integer algebra, pattern matching, regular expressions, quantum circuits, quantum simulation, binary decision diagrams.

Acknowledgment

I thank my Ph.D. advisor Tomáš Vojnar for bringing me in the magical land of formal methods 15 years ago, and Lukáš Holík for helping to create a vibrant (as well as fun) research environment in the VeriFIT group at FIT BUT. Further, I thank Vojta Havlena for being an amazing collaborator and apologize to his family for all the damage I have done to him. I thank Yu-Fang Chen for being my postdoc mentor, for inviting me numerous times to travel to the amazing beautiful country of Taiwan, and, last but not least, for introducing me to his former sister, who has later become my wife. I thank the members of the VeriFIT research group with whom I had the pleasure to share the crammed offices A219/A220 at FIT BUT (in alphabetical order): David Kozák, Filip Konečný, Filip Macák, Hanka Pluháčková/Šimková, Honza Fiedor, Jirka Matyáš, Jirka Pavela, Jirka Šimáček, Juraj Síč, Kamil Dudka, Lenka Turoňová/Holíková, Lukáš Charvát, Martin Hruška, Michal Rozsíval, Milan Češka jr., Ondra Valeš, Ondra Vašíček, Pavol Vargovčík, Petr Janků, Petr Muller, Roman Andriushchenko, Štefan Martiček, Tomáš Dacík, Tomáš Fiedor, Vendy Hrubá/Dudka, Veronika Šoková, Viktor Malík, and Zdeněk Letko. I also thank the other members of the VeriFIT group: Adam Rogalewicz, Aleš Smrčka, Milan Češka sr., Petr Peringer, and Slávek Křena. I cannot forget to thank my amazing students, who have, willingly, agreed to spend their time with me and my projects: Bára Šmahlíková, Honza Vašák, Jany Maťufka, Michal Hečko, Ondrej Alexaj, Sabína Gulčíková, Sára Jobranová, and Veronika Molnárová. In addition, I also thank other students that I have met along the way: Franta Nečas, Michal Kotoun, and Michal Šedý.

I thank my many collaborators (other than those that I have already mentioned above): Andrea Turrini, Anthony Widjaja Lin, Bengt Jonsson, Bow-Yaw Wang, Chiao Hsieh, Chih-Duo Hong, Cong Qui Trinh, Constantin Enea, David Chocholatý, Denis Matoušek, Fanda Blahoudek, Honza Kořenek, Ivan Homoliak, Jakub Semrič, Jie-Hong Roland Jiang, Jirka Matoušek, Juyn-Ao Lin, Kai-Min Chung, Lijun Zhang, Margus Veanes, Matthias Heizmann, Mihaela Sighireanu, Ming-Hsien Tsai, Olli Saarikivi, Parosh Aziz Abdulla, Peter Habermehl, Philipp Rümmer, Rupak Majumdar, Shin-Cheng Mu, Tony Tan, Tsung-Ju Lii, Wei-Lun Tsai, Yong Li, and Zhilin Wu; and friends from other institutions that have not been mentioned yet: Ahmed Bouajjani, Ahmed Rezine, Florian Zuleger, Honza Kofroň, Honza Strejček, Mohamed Faouzi Atig, Nicolas Mazzocchi, and Roland Meyer. I apologize for any omissions in the lists above.

Last but not least, I thank my parents, my sister with her family, and, above all, my wife and my daughter, for their support.

Over the time, I was supported by a number of grant projects, mainly by the Czech Science Foundation, the Ministry of Education, Youth and Sports of the Czech Republic, and the European Union.

Contents

Chapter 1

Introduction

This habilitation thesis concerns finite-state automata, a basic computational model known to every computer science undergraduate from their course on formal languages. Despite being such a simple model, finite-state automata have found their way into many applications in computer science, such as pattern matching, lexical analysis, controller design, software engineering, artificial intelligence, formal analysis and verification, synthesis, and hardware design to name only a few. Even now, the number of applications where finite-state automata are being used is growing, some of the applications enabled by the recent developments of more efficient algorithms for their manipulation. This thesis addresses two main objectives: (i) better algorithms for handling finite-state automata to improve the scalability of the automata technology and (ii) efficient use of the automata in applications, such as network monitoring, deciding logical theories, or analysing quantum programs.

Finite-state automata are a simple and elegant computational model with nice properties. Indeed, due to their elegance, some researchers (including yours truly) may joke about worshipping them with religious devotion. Although they have been introduced more than 80 years ago [169], many of their basic questions remain open. On the one hand, these are theoretical questions with far-reaching consequences—e.g., "*Can the* ***PSPACE****-complete nondeterministic finite automaton* (*NFA*) *universality problem be decided in polynomial time?*" On the other hand, there are many (more approachable) open problems regarding practical applications of the automata, such as how to efficiently complement them, test language inclusion, or even basic questions like how to efficiently test membership (potentially at speeds of hundreds of gigabits per second as in network intrusion detection systems, cf. Chapter 3).

Although finite-state automata are one of the most primitive computational models, having only constant space available for computation, many of their problems are, from a theoretical point of view, deemed infeasible. For instance, the universality and language inclusion problems are **PSPACE**-complete for NFAs and finite-state automata over infinite words (the so-called

Büchi automata, BAs) [94] and **EXPTIME**-complete for finite tree automata (TAs) [82]. Moreover, the complementation of NFAs and TAs with n states has a $\Omega(2^n)$ worst-case lower bound and the complementation of BAs has a $\Omega((0.76n)^n)$ worst-case lower bound [144, 231]. The minimization problems of these automata (i.e., finding an automaton with the same language and the smallest number of states) are also **PSPACE**-hard. In practical uses, even theoretically optimal algorithms whose complexity matches these lower bounds are unusable, unless significantly modified by the use of heuristics. Examples of these heuristics are the so-called *antichains* [10, 41, 86, 135, 158, 214, 230] for optimizing NFA/TA universality and language inclusion checking, or *simulation* and *bisimulation* relations [8, 9, 11, 12, 16, 55, 80, 108, 128, 134, 140, 186] for reducing the number of states of the automata (without guarantees of obtaining an automaton of the smallest size). These (and other) heuristics have made possible the use of automata in certain applications, such as regular model checking [1, 13, 42, 44], verification of programs with complex dynamic data structures [43, 115, 130, 132], or reactive synthesis [195].

The habilitation thesis contains a summary of the author's research in the area of development of efficient algorithms for handling finite-state automata over finite and infinite words and finite trees, and their use in applications.

First, the thesis focuses on improvements of complementation of finite-state automata over infinite words, the so-called *Büchi automata* (BAs). BAs (and related formal models, such as Rabin, Streett, Muller, or parity automata) are ubiquitous in reasoning over reactive systems—i.e., systems that operate indefinitely and react to incoming events, such as operating systems, controllers, arbiters, etc.—and in decision procedures of some logics (such as the monadic second-order logic of one successor S1S). Complementation of the automata is an operation that is essential in these applications. Although there are known algorithms whose complexity (in the number of generated states) matches the known lower bound of $\Omega((0.76n)^n)$, they can be applied in practice efficiently only if they are supplemented by heuristics. In Chapter 2, we will describe a number of such heuristics that helped to push the scalability of BA complementation to another level.

Next, we look at the problem of using classical finite automata (FAs) in high-speed *network intrusion detection systems* (NIDSes) using FPGA-based hardware accelerators in computer networks. One of the techniques used in NIDSes to detect malicious traffic is the so-called deep packet inspection, which checks whether the payload of a network packet matches a regular expression that describes an attack. The matching is usually performed using FAs (as the standard computational counterpart of regular expressions). The FAs for commonly used regular expressions are, however, too large to fit in the FPGA on the hardware accelerator to allow monitoring at the link speeds of up to 800 Gbps, and exact FA reductions are reaching their limits. Therefore, in Chapter 3, we will describe several approaches for *approximate* FA reductions that can be used to reduce the FAs so that we can fit enough of them on the FPGA in order to allow monitoring the network at the link speed with probabilistic guarantees on the introduced error (often negligible in practice).

The third topic this thesis focuses on is the *automata-logic* connection, a celebrated result (due to the founders of the field, such as Büchi [52, 53], Rabin [193], or Vardi and Wolper [221]) that shows that automata can be used in a natural way to reason in some logical theories. Examples of such theories are monadic second-order logics over natural numbers and trees (S1S, WS1S, S*k*S, WS*k*S), Presburger arithmetic, or flavours of linear temporal logic (LTL, QPTL, ...), which are all quite expressive and have numerous practical applications. The reasoning is performed by classical automata constructions (union, intersection, complement, projection). Performing these operations naively, however, usually does not scale—indeed, the logics often have a super-exponential (in some cases even **NONELEMENTARY**) worst-case lower bound. In order to have decision procedures that can work on practical examples, it is necessary to devise suitable heuristics. In Chapter 4, we present several of such heuristics for the logics WS1S, WS*k*S, S1S, and Presburger arithmetic. The heuristics helped to significantly advance the scalability of deciding the considered logics.

Another connection of automata with logic is addressed in Chapter 5, where we discuss the work on deciding the *theory of strings*. The theory of strings is an SMT theory that is used for reasoning about strings, usually in practical contexts, such as in program verification, symbolic execution, test generation, or even in analyses of resource access policies of cloud services [24, 201]. Currently, there exists a number of string solvers, based on different techniques, with different strengths and weaknesses, with none of them being clearly the best. In Chapter 5, we present two of our approaches to string solving, one based on regular model checking and one based on the so-called *stabilization* of systems of word equations, with both of these approaches using automata as their underlying formalism (in a completely different way though). The stabilization-based approach even made it into an industry-grade solver that won the strings category of the SMT-COMP'24 competition [211].

Finally, we will focus on a field that has recently attracted a lot of attention: *quantum computing*. Quantum computing promises solving many practical problems in physics, chemistry, pharmacology, etc., as well as some computational problems (e.g., factorization, unstructured search) much faster than classical computers. The current progress of the development of quantum computers gives us hope that there will be usable quantum computational hardware available in a not-so-distant future. Programming quantum computers is, however, several orders of magnitude more complex than classical programming, the discipline still being in its infancy with limited computer-aided support. Analysis of quantum programs will probably play even bigger role than in the case of classical programs. Existing tools for analyzing quantum programs are, however, usually quite limited in what they can do (e.g. simulation), highly manual (e.g. verification using interactive theorem provers), or do not scale. In Chapter 6, we first introduce a novel fully automated and efficient approach for deductive Floyd-Hoare-style verification of quantum circuits using finite tree automata and then a symbolic-execution-based approach for acceleration of loops in quantum programs, both of these contributions significantly improving scalability and/or expressiveness of automated analysis in quantum computing.

At the end of each chapter is the list of the author's publications contributing to the topic, with several representative papers attached in Appendix A.

Chapter 2

Büchi Automata Complementation

Nondeterministic Büchi automata (BAs) [52] are an elegant and conceptually simple framework to model infinite behaviors of systems and the properties they are expected to satisfy. The expressive power of BAs coincides with the class of the so-called *ω-regular languages*. BAs are widely used in many important verification tasks, such as termination analysis of programs [126], model checking [221], or as the underlying formal model of decision procedures for some logics (such as S1S [52] or a fragment of the first-order logic over Sturmian words [129]). Many of these applications require to perform *complementation* of BAs: For instance, in termination analysis of programs within Ultimate Automizer [126], complementation is used to keep track of the set of paths whose termination still needs to be proved. On the other hand, in model checking[1] and decision procedures of logics, complement is usually used to implement negation and quantifier alternation. Complementation is often the most difficult automata operation performed here; its worst-case state complexity is $\mathcal{O}((0.76n)^n)$ [15, 204] (which is tight [231]).

In contrast to the case of *nondeterministic finite automata* (NFAs), where the complementation algorithm is quite simple (it determinizes the NFA and swaps accepting and non-accepting states), the situation is more complex for BAs. The main difference is that a BA complementation algorithm cannot use standard determinization, since *deterministic* BAs have a strictly smaller expressive power than *nondeterministic* BAs. For instance, the ω-regular language given by the expression $(a + b)a^\omega$—which contains words over the alphabet $\{a, b\}$ with a finite number of occurrences of the symbol b—cannot be expressed by a deterministic BA.

Since the pioneering work of Büchi in 1962 [52], there has appeared a num-

[1]Here, we consider model checking w.r.t. a specification given in some more expressive logic, such as S1S [52], QPTL [208], or HyperLTL/HyperQPTL [79], rather than LTL [191], where negation is simple.

ber of approaches for complementing BAs. One of the often used approach is based on *determinization* into an automaton model whose deterministic subclass can still express all ω-regular languages, such as *Rabin automata* (the so-called Safra's construction [202]) or parity automata (the Safra-Piterman's construction [167, 190, 194]), and where the acceptance condition can be complemented (Rabin's condition can be easily complemented into Streett's condition, since the conditions are dual, and parity condition can be easily complemented by changing the parity condition—e.g., from *Min-Odd* to *Min-Even*—or by incrementing all labels by 1). Determinization-based complementation is, however, suboptimal (in theory and also often in practice). For instance, for the quite powerful subclass of *elevator automata* [124]—which are defined as BAs whose all maximum strongly connected components (SCCs) are either *inherently weak* (i.e., either all runs are accepting or all runs are rejecting) or *deterministic*—, determinization obtains in the worst case an automaton with $\Theta(n!)$ states [161], while their complementation can be done in $\mathcal{O}(4^n)$ [121], which is strictly better. The reason of this difference is that determinization imposes a stronger property on the output BA than complementation, which has more opportunities for optimizations.

Other than determinization-based algorithms, there have appeared several branches of Büchi complementation approaches: *Ramsey-based complementation*, was the very first complementation construction, where the language of an input automaton is decomposed into a finite number of equivalence classes. It was proposed by Büchi in [52] and was further enhanced in [49]. *Slice-based complementation* tracks the acceptance condition using a reduced abstraction on a run tree [145, 220]. *A learning-based approach* was introduced in [160, 162]. Allred and Ultes-Nitsche then presented a novel optimal complementation algorithm in [15]. For some special types of BAs, e.g., deterministic [155], inherently-weak [171], semi-deterministic [39], or unambiguous [98], there exist specific complementation algorithms with a better complexity. *Semi-determinization-based complementation* converts an input BA into a semi-deterministic BA [83], which is then complemented [37].

Rank-based complementation, studied in [104, 112, 146, 154, 204], extends the subset construction for determinization of finite automata by storing additional information in each macrostate to track the acceptance condition of all runs of the input automaton. Optimizations of an alternative (sub-optimal) rank-based construction from [154] going through *alternating Büchi automata* were presented in [112]. Furthermore, the work in [146] introduces an optimization of the algorithm from [204], in some cases producing smaller automata. In the rest of the chapter, we will describe our improvements of the rank-based construction, follow by optimizations of algorithms for specific subclasses of BAs, and conclude by an algorithm that combines several Büchi complementation procedures together.

2.1 Improving Rank-Based Büchi Complementation

Let us now focus on improvements of the rank-based complementation [104, 154, 204]. The rank-based construction produces a BA where states have an internal structure—we call them *macrostates*. In a macrostate, it stores a partial information about all runs of a BA $\mathcal{A}$ over some word w. In addition to tracking states that $\mathcal{A}$ can be in (which is sufficient, e.g., in the determinization of NFAs), a macrostate also stores a guess of the rank of each of the tracked states in the *run DAG* that captures all these runs. The guessed ranks impose restrictions on how the future of a state might look like (i.e., when $\mathcal{A}$ may accept). Although the rank-based construction can achieve the optimal $\mathcal{O}((0.76n)^n)$ (modulo a quadratic factor) worst-case state complexity [204], the performance of the basic algorithm is in many practical cases insufficient. In our work, we have introduced several optimizations that exploit the structure of the rank-based algorithm and generate, in practice, much smaller output BAs.

First, in [70], we improved the optimal rank-based procedure that started in the works of Kupferman, Friedgut, and Vardi [104, 154] and achieved the theoretically optimal complexity in the work of Schewe [204]. We improved the procedure by exploiting *simulation* between states. Recall that a simulation is a binary relation over states of a BA that underapproximates trace behaviour. For instance, if a *direct* simulation holds between states p and q, written as $p \preceq_{di} q$, it holds that for every trace τ_p leaving p, there exists a trace τ_q leaving q over the same word such that at every step where there is an accepting state on τ_p, there will also be an accepting state on τ_q too. Likewise, if a *delayed* simulation holds between p and q, i.e., $p \preceq_{de} q$, then for every trace τ_p leaving p, there exists a trace τ_q leaving q over the same word such that if there is an accepting state on τ_p at step s, there will also be an accepting state on τ_q at a step $s' \geq s$.

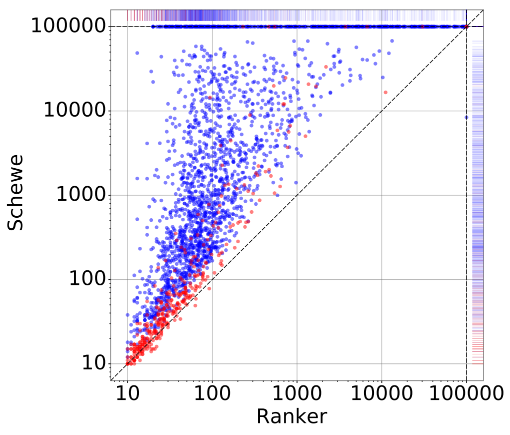
In the rank-based complementation algorithm of [204], macrostates of the constructed complement BA have the following structure: (S, O, f, i) where $S, O \subseteq Q$ (with Q being the set of states of the original BA of the size n), f is a *ranking function* of the type $f \colon Q \to \{0, \ldots, 2n-1\}$ with some restrictions[2], and $i \in \{0, 2, \ldots, 2n-2\}$. The work in [70] introduces two classes of optimizations of the algorithm based on the use of simulation. The first class of optimizations prunes the constructed macrostates. In particular, we have shown that if we reach a macrostate $m = (S, O, f, i)$ such that there are states $p, q \in S$ with $p \preceq_{di} q$ and $f(p) > f(q)$, we can safely remove m from the output BA. Intuitively, $f(p) > f(q)$ in some sense says that p can have "*more behaviours*" than q, while $p \preceq_{di} q$ says that p can have "*less behaviours*" than q, and these two properties are incompatible. Similarly, one can also remove m if $p \preceq_{de} q$ and $f(p) > \llbracket f(q) \rrbracket$ where $\llbracket i \rrbracket$ for $i \in \mathbb{N}$ denotes the smallest even

[2]In particular, the restrictions are the following: (i) f maps accepting states to even numbers, (ii) $\max\{f(q) \mid q \in Q\} = r$ is odd, and (iii) $\{f(q) \mid q \in Q\} \supseteq \{1, 3, \ldots, r\}$.

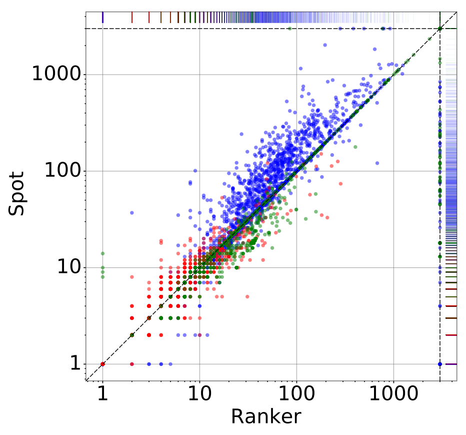
number $\geq i$. The second class of optimizations in [70] is based on *saturation* of macrostates: we can saturate the S-component of every macrostate to $cl[S] = \{q \in Q \mid \exists s \in S \colon q \preceq_{de} q\}$. This can give us a smaller number of total states, as some macrostates will be saturated to larger macrostates that would be present anyway (but it can also increase the total number of states since, generally speaking, macrostates with more states can generate more successors, because for more states, there are also more ranking functions).

We further improved the rank-based complementation in [120] by several optimizations that use the structure of the input BA $\mathcal{A}$ and the structure of the subset automaton $\mathcal{R}_{\mathcal{A}}$ obtained from $\mathcal{A}$ by the classical subset construction (as for the determinization of NFAs). Generally speaking, the optimizations are trying to reduce the number of considered ranking functions f, which is the dominating factor of the state complexity. We achieve this by exploiting the properties of the so-called *tight* rankings (first introduced in [104]). For instance, for a macrostate (S, O, f, i), the maximum rank of f is bounded by $2|S| - 1$. The construction, however, demands that in the resulting automaton it holds that on any run, the maximum rank of every macrostate is the same. Therefore, we can conclude that the maximum rank of f that we need to consider is bounded by $2m - 1$ where m is the maximum size of a macrostate of $\mathcal{R}_{\mathcal{A}}$ that is in a non-trivial SCC of $\mathcal{R}_{\mathcal{A}}$ and is reachable in $\mathcal{R}_{\mathcal{A}}$ from S. The previous can be elaborated on by doing a more refined analysis, taking into consideration also final states, etc., to achieve an even better reduction in the considered ranks for the states. The proposed optimizations, however, do not work with the algorithm from [204] as is, so we needed to modify it; details are rather technical.

In the follow-up paper [124], we came up with more radical approaches for decreasing the maximum ranks considered in the rank-based complementation procedure. First, we further improved the techniques from [120] by introducing a fixpoint-based algorithm for analysing maximum ranks induced by macrostates in SCCs of $\mathcal{R}_{\mathcal{A}}$. The algorithm keeps track of how the ranks of states within a macrostate can flow, often obtaining a much finer estimate of the maximum needed ranks quickly (the problem of computing the true maximum needed ranks is **PSPACE**-complete [112]). The second approach for decreasing the considered ranks we introduced is based on an idea that is crucial for efficient algorithms (more on that later): looking at the structure of the input BA $\mathcal{A}$, in particular into the types of its SCCs. We found out that for some types of SCCs—namely deterministic and inherently weak—there is a limit on the increase of the maximum rank that needs to be considered (between 0 and 2 based on the way how the SCCs are connected) that is independent of the number of states in the SCC. Inspired by this observation, we defined the class of *elevator automata*—BAs that contain only deterministic and inherently weak SCCs—, which can be complemented using the rank-based algorithm in $\mathcal{O}(16^n)$ (this was later improved in [121] for a non-rank-based algorithm). The class of elevator automata is quite expressive (it includes



(a) Ranker vs. Schewe



(b) Ranker vs. Spot

Figure 2.1: (a) Comparison of the number of states generated by a vanilla rank-based complementation algorithm with the optimal theoretical complexity from [204] (denoted as Schewe) and using our optimizations (denoted as Ranker).
(b) Comparison of the number of states (after postprocessing) for complement automata constructed by Ranker and Spot.
Axes are logarithmic. Blue data points correspond to randomly generated BAs, red data points correspond to BAs obtained by translation of LTL formulae, and green data points correspond to BAs obtained from Ultimate Automizer (only in (b)).

semi-deterministic BAs [83] and, e.g., in a standard benchmark of BAs obtained from LTL formulae, around 90 % of the BAs are elevator automata) and can be exploited by algorithms other than complementation; e.g., in [161], the authors gave a better upper bound on the complexity of determinization of elevator automata compared to general BAs.

We implemented the described (and many more) heuristics in a mature tool named Ranker [122]. In Figure 2.1a, we show a comparison of the generated state space compared to the vanilla rank-based complementation algorithm with the optimal theoretical complexity from [204]. The savings in the generated state space are indeed dramatic (we note that the state space generated by the algorithm in [204] is still much smaller than that of the previous rank-based algorithms from [104, 154]). Not only did Ranker improve the rank-based algorithms, it is also highly competitive when compared to other state-of-the-art algorithms, including determinization-based algorithms [190, 194, 202] implemented in Spot [87], often considered the most sophisticated tool for handling Büchi automata (and other types of ω-automata) as of the date. Figure 2.1b shows that Ranker could often obtain much smaller complements than Spot, and the average size of the obtained automaton was smaller.

2.2 Complementation of Special Classes of BAs

As mentioned above, considering the types of BAs (or its SCCs) is crucial for efficient BA complementation algorithms, since many BAs occurring in practice have a structure that can be exploited. For instance, deterministic BAs with n states can be complemented to BAs with at most $2n$ states [155], inherently weak BAs can be complemented to BAs with no more than $\frac{2}{3}3^n$ states [171] (the lower bound is 2^n), and semi-deterministic BAs (i.e., BAs that are deterministic after seeing an accepting state for the first time) can be complemented to BAs with at most 4^n states [39].

These constructions can, however, still be optimized be perform better in practice. For instance, in [73], we proposed optimizations of the NCSB construction from [39] that made the procedure generate less states in many practical benchmarks (in particular, the work was done in the context of termination checking of C programs by Ultimate Automizer [126], which uses BA complementation within its procedure). Among others, the paper defined two new types of simulations, *early* simulation ($\preceq_e$) and *early*+1 ($\preceq_{e+1}$) simulation, as follows. If $p \preceq_{e+1} q$, then for every trace τ_p leaving p, there exists a trace τ_q leaving q over the same word such that if there are two accepting states on τ_p at steps i and j, with $i < j$, then there is an accepting state on τ_q at step k such that $i < k \leq j$. Intuitively, this means that between every two accepting states on τ_p, where is an accepting state on τ_q. The definition for $p \preceq_e q$ is similar with the addition that there needs to be an accepting state on τ_q even before the first occurrence of an accepting state on τ_p. Likewise to other simulations, early and early+1 simulations imply language inclusion. While we are currently not aware of an algorithm for computing maximum relations that are $\preceq_e$ and $\preceq_{e+1}$ on a BA (studying the relation in detail is an interesting future work with a hope that it might be easier to compute than delayed and fair simulations—where the best currently known algorithm [95] we are aware of has the $\mathcal{O}(mn^3)$ time complexity—and be similarly rich or even richer), the simulation plays an important role in the complementation and inclusion checking of certain classes of BAs.

The reason why we came up with the early and early+1 simulations is that they can be easily computed[3] on the structure of macrostates that occur in certain BA complementation procedures, without the need to have the whole automaton constructed (which one needs for the standard—i.e., direct, delayed, and fair—simulations). For the use in complementation, if we combine a complementation algorithm with an algorithm for testing emptiness of the language of a macrostate (in order to directly prune out useless part of the result), we can keep a set *Empty* of macrostates that are known to have an empty language and when we encounter a macrostate M such that there

[3]More precisely, there exist non-trivial, often quite rich, relations that can be easily computed that *are* early or early+1 simulations, in other words, that *underapproximate* the maximum early or early+1 simulations, as they are not necessarily maximum.

exists $M' \in \mathit{Empty}$ with $M \preceq_e M'$ or $M \preceq_{e+1} M'$, then we do not need to explore M and can directly conclude that $\mathcal{L}(M) = \emptyset$ (this is an idea in a similar family of techniques as the subsumption used in the *Antichains* algorithm for NFA universality and inclusion checking [10, 230]).

In [73], the optimized complementation mentioned above is used in the context of computing language difference (where complementation is a subprocedure) of semi-deterministic BAs using the NCSB algorithm [39]. We define subsumption relations $\sqsubseteq$ and $\sqsubseteq^B$ on the macrostates constructed in the NCSB algorithm. To give an idea (without going into technical details, which can be found in [39, 73]), a macrostate of NCSB has the structure (N, C, S, B), where $N, C, S, B \subseteq Q$ are sets of states of the input BA $\mathcal{A}$. The two subsumption relations are defined as follows:

$$(N_1, C_1, S_1, B_1) \sqsubseteq (N_2, C_2, S_2, B_2) \Leftrightarrow N_1 \supseteq N_2 \wedge C_1 \supseteq C_2 \wedge S_1 \supseteq S_2 \quad \text{and}$$
$$(N_1, C_1, S_1, B_1) \sqsubseteq^B (N_2, C_2, S_2, B_2) \Leftrightarrow N_1 \supseteq N_2 \wedge C_1 \supseteq C_2 \wedge S_1 \supseteq S_2 \wedge B_1 \supseteq B_2.$$

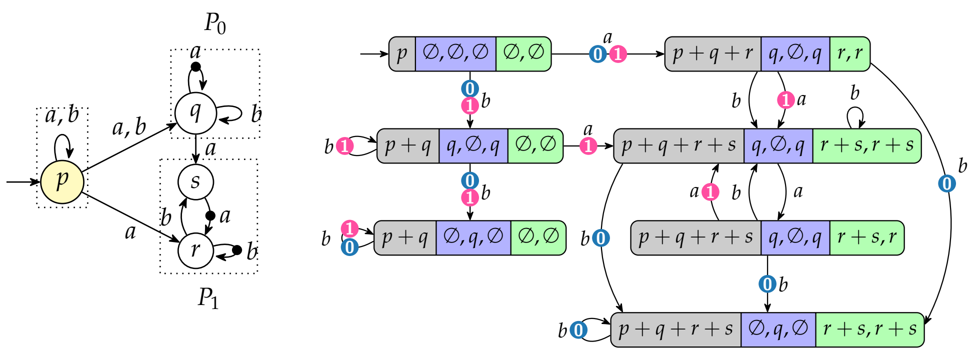
Notice that the relations can be computed just from the structure of the macrostates, without any knowledge about the structure of the complement BA (which is, however, determined by the macrostates). In [73], we prove that $\sqsubseteq$ is an early+1 simulation and that $\sqsubseteq^B$ is an early simulation, so they can be used for the optimizations. First steps for using the subsumption relations also in the context of checking language inclusion has already been made in [14], but a proper evaluation is a part of the future work.

Regarding other types of BAs, we have also improved algorithms for complementation of inherently weak BAs and semi-deterministic BAs in [122]. For inherently weak BAs, we propose a way of saturating/pruning macrostates using a simulation relation in the algorithm from [171] in a similar fashion as in [70, 108] and for semi-deterministic BAs, we propose a modification of the NCSB algorithm from [39] using a technique to limit the number of outgoing edges of a macrostate in a similar way as in the algorithms from [120, 204].

2.3 Complementation of BAs via SCC Decomposition

Our most recent work on BA complementation gets inspiration from the treatment of SCCs in the input BA based on their type from [124] and the SCC decomposition-based determinization algorithm from [161]. The approach, published in [121], is based on the main idea that in the input BA $\mathcal{A}$, every infinite run eventually stays in some SCC. One can, therefore, consider runs for every SCC independently[4]. Then, it is possible to exploit the particular

[4]One could, actually, transform the input BA $\mathcal{A}$ into a union of k copies of $\mathcal{A}$, one for each of the k SCCs, such that each copy keeps accepting states only in one SCC. This, however, disallows sharing of common states.



(a) A BA $\mathcal{A}_{ex}$ (b) A complement of $\mathcal{A}_{ex}$

Figure 2.2: An example of complementing a BA (with transition-based acceptance) using the SCC decomposition-based algorithm from [121]. Note that $\mathcal{A}_{ex}$ has two accepting SCCs: P_0 (deterministic) and P_1 (inherently weak). States in the complement are given as $(H, (C_0, S_0, B_0), (C_1, B_1))$ where H tracks all runs in $\mathcal{A}$, (C_0, S_0, B_0) is a partial macrostate for the [39]-inspired complementation of deterministic SCCs, and (C_1, B_1) is a partial macrostate for the [171]-inspired complementation of inherently weak SCCs; to avoid too many braces, sets are given as sums. The acceptance condition of the complement is $\mathsf{Inf}(⓿) \wedge \mathsf{Inf}(❶)$.

classes of the SCCs, even when $\mathcal{A}$ as a whole does not fall into one particular class. For each of the SCCs, we can use a different algorithm that is optimized for the type of the given SCC. For instance, for SCCs with no accepting state, we can just use an NFA-like determinization that needs at most 2^n macrostates (with n being the number of states *in the SCC* this time), for accepting inherently weak SCCs, we can use a modification of the algorithm from [171] bounded by $\frac{2}{3}3^n$, for deterministic SCCs, we can use a customization of some flavour of the NCSB algorithms [39, 73, 122] bounded by 4^n, and for general nondeterministic SCCs, we can use a modification of one of the many standard BA complementation algorithms (which can be restricted to the given SCC, e.g., for rank-based algorithms, we can restrict the maximum rank only by the number of states in the SCC and not in the whole $\mathcal{A}$). One then needs a top-level algorithm that orchestrates the run of the SCC-local algorithms. Using this combination of several complementation procedures, we can use a naturally occurring richer acceptance condition in the complement—in particular, since the algorithms of all SCCs need to accept, one gets a conjunction of the conditions for the partial algorithms. The partial algorithms can then use any acceptance condition; e.g., the so-called *initial deterministic components* (deterministic components into which one can only arrive deterministically) can be easily complemented using a co-Büchi (Fin) condition (in general, the output of our SCC-based complementation algorithm is an Emerson-Lei automaton [23, 90], i.e., an ω-automaton where the acceptance condition is a Boolean combination of Inf conditions). See Figure 2.2 for an example of the output of our complementation algorithm.

A tantalizing feature of this approach is that one has a general framework into which they can connect new algorithms specialized for certain types of SCCs in a plug-and-play fashion. For instance, we are currently investigating specialized algorithms for deterministic components with more general acceptance conditions, e.g., generalized, Streett, Emerson-Lei, etc. These algorithms can then be applied only for the particular SCC and they are almost always more efficient than the currently often used algorithms that first simplify the acceptance condition (which usually increases the size of the BA considerably) and then run a standard algorithm.

Contributed Papers[5]

[70] Yu-Fang Chen, Vojtěch Havlena, and Ondřej Lengál. "Simulations in Rank-Based Büchi Automata Complementation". In: *Proc. of APLAS'19*. Vol. 11893. LNCS. Springer, 2019, pp. 447–467.

[73] **Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. "Advanced automata-based algorithms for program termination checking". In: *Proc. of PLDI'18*. ACM, 2018, pp. 135–150.**

[120] Vojtěch Havlena and Ondřej Lengál. "Reducing (To) the Ranks: Efficient Rank-Based Büchi Automata Complementation". In: *Proc. of CONCUR'21*. Vol. 203. LIPIcs. Dagstuhl, 2021, 2:1–2:19.

[121] **Vojtěch Havlena, Ondřej Lengál, Yong Li, Barbora Šmahlíková, and Andrea Turrini. "Modular Mix-and-Match Complementation of Büchi Automata". In: *Proc. of TACAS'23*. Vol. 13993. LNCS. Springer, 2023, pp. 249–270.**

[122] Vojtěch Havlena, Ondřej Lengál, and Barbora Šmahlíková. "Complementing Büchi Automata with Ranker". In: *Proc. of CAV'22*. Vol. 13372. LNCS. Springer, 2022, pp. 188–201.

[124] **Vojtěch Havlena, Ondřej Lengál, and Barbora Šmahlíková. "Sky Is Not the Limit - Tighter Rank Bounds for Elevator Automata in Büchi Automata Complementation". In: *Proc. of TACAS'22*. Vol. 13244. LNCS. Springer, 2022, pp. 118–136.**

[5]The works [73, 121, 124] (in bold) are attached to this thesis.

Chapter 3

Finite Automata in Network Intrusion Detection

Pattern matching of text using regular expressions is ubiquitous in many areas of information technology. The speed of the pattern matching can be crucial in some applications. An instance of such an application is the so-called *deep packet inspection* within *network intrusion detection systems*, which tries to find potentially malicious patterns in the payload of packets entering the network. As the speed of current computer networks grows quickly (with backbones and entry points running on speeds up to 100/200/400/800 Gbps), single-machine software-only solutions cannot follow (they can scale up to only units of Gbps). Therefore, one either needs to parallelize the task (e.g., by having a cluster of dozens to hundreds of computers doing pattern matching of one network link) or use a hardware accelerator, which we focus on here. Since the patterns one needs to match often change in time (e.g., when a new vulnerability in some software is detected and its exploits start appearing in live traffic), the hardware accelerator needs to support reconfiguring the pattern bank. One approach, taken, e.g., in [168], is to use reconfigurable hardware in the form of *field-programmable gate arrays* (FPGAs), where the patterns are represented using NFAs, with each state of the NFA being assigned a one-bit register (a flip-flop) on the chip. The use of NFAs allows a usually much more compact encoding when compared to *deterministic finite automata* (DFAs), which can be up to exponentially larger (and in practice are indeed often significantly bigger). Apart from software pattern matching with NFAs—which usually needs to perform backtracking in order to try all possible runs of the NFA over the input word—, in hardware, since every state is represented by a register, no backtracking is necessary, because all possibly reachable states can have their registers active at the same time.

Despite the compact encoding achieved by using NFAs, keeping the NFAs as small as possible is essential, as hardware resources are expensive (and often limited by the chip used on a multi-purpose FPGA-based network accelerator). For real-world sets of patterns used in intrusion detection (e.g., those provided within Snort rules [213]), even after reducing the size of the NFA using standard language-preserving reduction techniques (e.g., using the (bi-)simulation-based reduction [55, 140, 186]), the size of the resulting NFA is too large to fit on the chip. A solution is to use the hardware as an approximate pre-filter that discards the majority of packets that do not match the attack patterns (the amount of traffic with attacks of such a kind is usually negligible) and sends the rest (usually a tiny fraction) to software to be matched exactly. A challenging part of this problem is how to perform the approximation in a good way. Sampling the input does not work since we could then drop a malicious packet (usually, only false positives—i.e., over-approximation of the set of matched packets—are allowed).

To address this issue, in [59, 60], we propose an *approximate* NFA reduction, i.e., a reduction that does not preserve the language of the NFA. Our reduction changes the NFA $\mathcal{A}$'s language, but in a controlled manner, with probabilistic guarantees w.r.t. a probability distribution over the set of input words (i.e., network packets). The probability distribution is represented by a *probabilistic automaton* (PA), an NFA whose transitions are labelled with probabilities and states are given probabilities of accepting. The PA gives every word $w \in \Sigma^*$ its probability $Pr(w) \in \langle 0, 1 \rangle$ such that $\sum_{w \in \Sigma^*} Pr(w) = 1$. We constructed such a PA for our use case by creating a DFA representing the general structure of the HTTP protocol (we focused on attacks carried out over HTTP) and then passing a large amount of packets from real-world network traffic (obtained from collection points of a backbone network) through it, learning probabilities of transitions (see [60] for a discussion about why using some general probabilistic automaton learning approach, such as the Alergia algorithm [57], did not work).

We proposed the following two kinds of reductions:

1. *size-driven reduction*, which asks about the NFA $\mathcal{B}$ with n states such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{L}(\mathcal{A})$ and $\sum_{w \in \mathcal{L}(\mathcal{B}) \setminus \mathcal{L}(\mathcal{A})} Pr(w)$ is the smallest (i.e., the probability of misclassifying an incoming packet is as small as possible), and

2. *error-driven reduction*, where we are given the maximum allowed error e and need to construct an NFA with the least number of states that over-approximates the language of $\mathcal{A}$ and whose probability of misclassifying a packet is bounded by e.

We show that optimal solutions to both problems are as hard as NFA minimization [143] (i.e., **PSPACE**-complete) and devise greedy algorithms that identify states with the lowest probability to be reached and remove them from the automaton. While these algorithms give sub-optimal results, they can be run in a reasonable time on the input (quite large) NFAs and, in practice, they produce results that are good enough. For instance, for the given FPGA-based accelerator that we used, without the approximate reduction, we could monitor the patterns from the `http-backdoor` set of Snort rules only at the speed of 38.4 Gbps. With the approximate reduction, we could fit more matching units into the FPGA and monitor the rules at the link speed of 100 Gbps with the negligible error of 3.4×10^{-8} w.r.t. the probability distribution. On the other hand, for the patterns from the `http-malicious` rule set, without approximate reduction, the matching could be done at the speed of 249.6 Gbps, and with our approximate reduction, we could achieve the link speed of 400 Gbps with the error of 8.7×10^{-8}.

In [58], we further extended the work in two ways:

1. We built on the idea of a *pre-filter* and introduced a hardware architecture that uses several stages where the matching happens, each stage being more precise than the previous stage while having a lower throughput. The idea is that in the given hardware architecture, e.g., for the input bandwidth of 100 Gbps, one needs to use 64 parallel matching units, each with the throughput of 1.6 Gbps. The size of one unit is, however, too large for 64 of them to fit on the chip. Instead of that, we can use 64 much smaller approximate units that decrease the bandwidth to, say, 3 Gbps, and follow the first stage by another stage that can contain two exact matching units, or, if this would still be too large, another approximate, but more precise, matching units, etc.

2. The use of a probabilistic automaton gives a compact representation of the network traffic but comes with the disadvantage that computation of probabilities of states is quite computationally intensive. Instead of using such a probabilistic automaton, in [58] we use directly the network traffic sample to check which states in the NFA are less likely than other states to be visited during matching of a packet.

With this technique, we achieved exact matching on the `http-backdoor` Snort rule set at the speed of 400 Gbps and on the much more challenging `http-spyware` Snort rule set, we achieved exact matching at the speed of 200 Gbps and approximate matching at the speed of 400 Gbps with the error of 4 % (compared to the speed of 17 Gbps of exact matching when using the standard approach).

Contributed Papers[1]

[58] Milan Češka, Vojtěch Havlena, Lukáš Holík, Jan Kořenek, Ondřej Lengál, Denis Matoušek, Jiří Matoušek, Jakub Semrič, and Tomáš Vojnar. "Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata". In: *Proc. of FCCM'19*. IEEE, 2019, pp. 109–117.

[1]The work [60] (in bold) is attached to this thesis.

[59] Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection". In: *Proc. of TACAS'18*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. LNCS. Springer, 2018, pp. 155–175.

[60] **Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Approximate reduction of finite automata for high-speed network intrusion detection". In: *Int. J. Softw. Tools Technol. Transf.* 22.5 (2020), pp. 523–539.**

Chapter 4

The Automata-Logic Connection

One of the possible uses of the automata technology developed in Chapter 2 is as a backend of decision procedures for certain logics. Let us give examples of well-known logics that have automata-based decision procedures based on the type of automata used:

- NFAs: the weak monadic second-order logic of 1 successor (WS1S) [53, 88, 109], the monadic second-order logic over finite strings [127], the Presburger arithmetic [45, 229], and the Büchi arithmetic [50, 53];
- BAs: the (full) monadic second-order logic of 1 successor (S1S) [52], the first-order theory of Sturmian words over Presburger arithmetic [129], and linear-time temporal logics, like LTL [221], QPTL [208], or Hyper-LTL/HyperQPTL [79];
- tree automata [82]: the (full/weak) monadic second-order logic of k successors (SkS/WSkS) [88, 193, 212].

In addition, some other logics have decision procedures that translate to the mentioned logics, e.g., some fragments of separation logic [141].

In this chapter, we will first describe our contributions to improving the scalability of deciding WS1S/WSkS/S1S formulae and then give an overview of techniques that make an automata-based decision procedure for Presburger arithmetic competitive (and often faster) to other approaches that are currently prevalently used within state-of-the-art SMT solvers.

4.1 Deciding Monadic Second-Order Logics

First, we will focus on the monadic second order logics, in particular the weak monadic second-order logic of $1/k$ successor(s): WS1S/WSkS. These are classical monadic second-order logics, i.e., quantified variables range over *sets* of elements of the universe, but they are *weak*, which means that the considered sets are *finite*. Therefore, e.g., the formula $\psi\colon \exists R \forall S\colon S \subseteq R$ does not hold in these logics (as opposed to their non-weak siblings S1S/S*k*S, where quantifiers range over *all* sets; the formula ψ holds in those).

Let us start with defining a minimal syntax of a formula φ in WS1S:

$$\begin{aligned} \varphi_{atom} &::= X \subseteq Y \mid X = Y + 1 \\ \varphi &::= \varphi_{atom} \mid \varphi \land \varphi \mid \neg\varphi \mid \exists X\colon \varphi \end{aligned} \tag{4.1}$$

where X, Y are second-order variables. The semantics of $X \subseteq Y$ is standard, the formula $X = Y + 1$ holds iff $X = \{y + 1 \mid y \in Y\}$, and $\exists X$ quantifies existentially over finite subsets of the universe. In the case of WS1S, the considered universe is the set of natural numbers $\mathbb{N}$. The syntax can be easily extended to WS*k*S, for a fixed natural number $k > 1$ by substituting the successor function formula $X = Y + 1$ by k formulae $X = S_1(Y), \ldots, X = S_k(Y)$, where S_i's denote different successor functions (for WS1S, there was just one denoted as $+1$). WS*k*S is then interpreted over the set of positions in an infinite k-ary tree (see Figure 4.2 later).

Although the logics have a simple syntax and are decidable, their decision problem complexity is **NONELEMENTARY** [170] (more precisely, the problem is complete for the **TOWER** complexity class [205], which is at the lower border of **NONELEMENTARY**).

The standard decision procedures for these logics are based on automata. The core of the decision procedures is simple:

(i) for each atomic formula, we construct an automaton representing exactly all its models using a suitable encoding of models into words or trees;

(ii) for formulae representing logical operations, we perform standard automata manipulation: for conjunction, we perform automata intersection (potentially preceded by cylindrification in the case the alphabets do not match), for negation, we perform automata complement, and for existential quantification, we perform projection on the alphabet of the automation followed by saturation of final states; and

(iii) in the end, we test language emptiness of the obtained automaton: if the language is non-empty, the formula is satisfiable.

The automata considered depend on the logic: the procedures use NFAs for WS1S and tree automata for WS*k*S.

Despite the forbidding theoretical worst-case complexity of the logics, their properties (great expressivity while being decidable) prompted the development of solvers that would be able to solve real-world formulae (e.g., formulae coming from HW verification [30], controller synthesis [46], reasoning about distributed reactive systems [152, 153], computational linguistics [175], or program verification [89, 142, 173]) in a practical time. This started with the implementation of [109] and later achieved practicality with the tools Mona [127] and Mosel [147]. The success of the tools is based on various optimizations, both high-level (such as developing a new formal model, the so-called *guided tree automata* [35], DAG-based representation of the formula [151], or the three-valued acceptance for automata [150]) and low-level (such as the use of *multi-terminal binary decision diagrams* (MTBDDs) [25, 51, 105] for representing the automata's transition functions and the use of cache-conscious data structures [151]). Although these implementations often work well, there are practical cases of formulae when the complexity strikes back and the tools fail.

One reason for these failures is that the tools are usually (in particular both Mona and Mosel) based on *deterministic* finite automata (DFAs), which are often exponentially larger than their nondeterministic counterparts (NFAs). In our previous work [102, 103], we have already shown how one could lift the concept of *antichain*-based universality testing of NFAs [10, 230] to satisfiability testing of WS1S formulae and avoid explicit determinization.

Here, we will present several works that significantly pushed the borders of practically decidable formulae of WS1S/WS*k*S. In [101], we introduced the so-called *lazy automata techniques* in the context of deciding WS1S. These techniques are based on looking at the structure of the states of the NFA for the input formula φ. One can see

1. states of NFAs for atomic formulae as atoms,
2. states of NFAs for formulae $\varphi_1 \wedge \varphi_2$ as pairs (q_1, q_2) where q_1 is a state of the NFA for φ_1 and q_2 is a state of the NFA for φ_2,
3. states of NFAs for formulae $\neg\varphi$ as sets of states $\{q_1, \ldots, q_n\}$ where q_1, $\ldots$, q_n are states of the NFA for φ, and
4. states of NFAs for formulae $\exists X\colon \varphi$ as states q of the NFA for φ.

For instance, for the formula

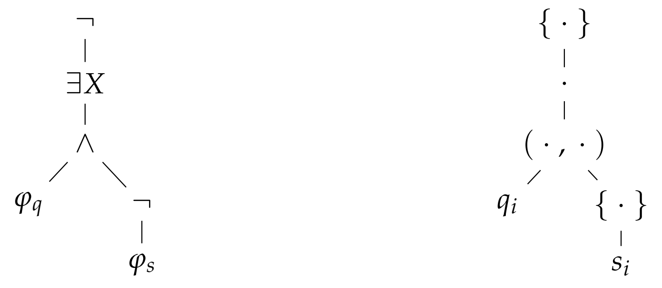
$$\neg\exists X\colon \varphi_q \wedge \neg\varphi_s \tag{4.2}$$

where φ_q and φ_s are atomic formulae, an example of a state of the corresponding NFA might look like the following:

$$\{(q_1, \{s_1, s_2\}), (q_2, \{s_1, s_3\}), (q_1, \{s_1, s_3, s_4\})\} \tag{4.3}$$

where q_i states are from the NFA for φ_q and s_i states are from the NFA for φ_s. See Figure 4.1 for an illustration of how the structure of a state corresponds to the abstract syntax tree (AST) of the formula.

We can then solve some questions about the NFA for the input formula φ without constructing NFAs for all sub-formulae of φ, *lazily*. This means that



(a) The AST of the example formula (b) The structure of a state of the NFA

Figure 4.1: The AST of the example formula and the structure of the state of the corresponding NFA.

states of the NFA will be constructed only on-demand during, e.g., performing a traversal through the NFA when searching for an accepting state in the context of testing non-emptiness of the NFA (i.e., testing satisfiability of φ). This can sometimes significantly speed up the decision procedure, e.g., when testing satisfiability of a formula $\varphi_U \wedge \varphi_B$ where φ_U is a small unsatisfiable formula and φ_B is a big formula, since, in such a case, we can completely avoid constructing the NFA for φ_B.

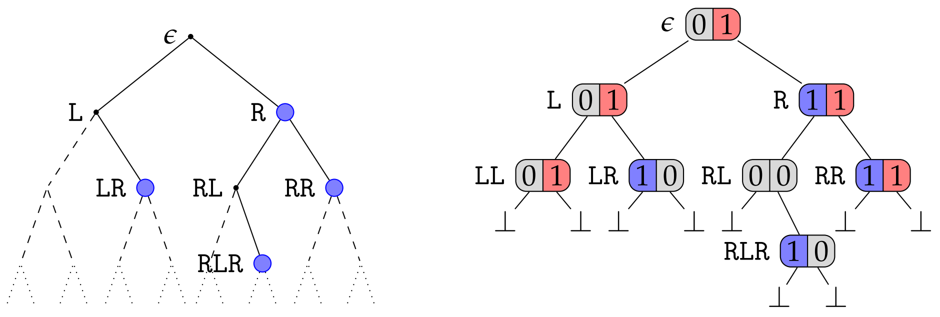
We formally describe the lazy procedure as an evaluation of a term constructed from the structure of the input formula and the atomic NFAs. For the example formula above (assuming that it contains no free variables), the corresponding term would be the following:

$$t = \pi_X\overline{(I_q - \{[X\colon 0],[X\colon 1]\}^* \wedge \overline{I_s - \{[X\colon 0],[X\colon 1]\}^*})} \tag{4.4}$$

where $-U$ for a set of words U denotes the language derivative w.r.t. U, I_q and I_s are the sets of initial states of the NFAs for φ_q and φ_s respectively, π_X denotes projection of a language by variable X, and $[X\colon c]$ denotes a symbol that maps the variable X to the Boolean value $c \in \{0,1\}$. Testing satisfiability of the input formula can be done by checking whether $\epsilon \in t$ (recall that we assume that the formula contains no free variables) by evaluating the term step by step using rewriting rules that unfold the derivatives.

In addition to the lazy approach, we also introduced subsumptions that can be used to simplify some of the obtained terms based on a similar idea as the antichains algorithm for testing NFA universality and inclusion [10, 230]. The subsumption is defined inductively on the structure of the states (i.e., it is not a simple set inclusion any more).

The lazy automata technique were further extended in [118, 119] to WSkS, i.e., extended from NFAs to tree automata. Recall that WSkS is interpreted over positions in the infinite k-ary tree, as depicted in Figure 4.2. Therefore, one cannot use simple language derivatives as in the case of WS1S, since when



(a) Positions assigned to the variable X (b) Encoding of ν into a tree τ_ν.

Figure 4.2: An example of an assignment ν to a pair of variables $\{X, Y\}$ such that $\nu(X) = \{\mathtt{LR}, \mathtt{R}, \mathtt{RLR}, \mathtt{RR}\}$ and $\nu(Y) = \{\epsilon, \mathtt{L}, \mathtt{LL}, \mathtt{R}, \mathtt{RR}\}$ and its encoding into a tree τ_ν. A node at a position p has the value $\boxed{x \mid y}$ where $x = 1$ if and only if $\tau_\nu(p)$ maps X to 1 and $y = 1$ if and only if $\tau_\nu(p)$ maps Y to 1.

treating trees, one needs to, in some cases, remember the *context*. For instance, given a set of states, when reasoning about trees rooted in the states from the set, and talking about their left and right subtrees, one needs to keep track of the fact which left subtree corresponds to which right subtree. We then extended the notions of term evaluation and subsumption for terms in WS1S from [101] to the terms in the WS*k*S procedure, which talk about *tree derivatives*.

During the work on optimizations of the core WS1S/WS*k*S decision procedures (papers [101–103, 118, 119]), we noticed that the particular form of the input formula can play a big role in the efficiency of the underlying decision procedure. In [117], we explored this idea further by introducing several rewriting rules for WS*k*S formulae that could significantly decrease the size of its state space. The rules are mainly based on the notion of *antiprenexing*, i.e., pushing quantifiers *inside* of a formulae (as opposed to prenexing, where quantifiers are moved to the outmost scope). This usually helps because the lower a quantifier is, the smaller is the size of the alphabet one needs to work with. There are rules that are almost always advantageous to use, e.g., applying distribution of the existential quantifier over disjunction

$$\exists X\colon \varphi \lor \psi \quad \rightsquigarrow \quad (\exists X\colon \varphi) \lor (\exists X\colon \psi), \tag{4.5}$$

some rules, however, help only sometimes. For instance, it is not always clear whether one should use the following distributive law:

$$\exists X\colon (\varphi \lor \psi) \land \chi \quad \rightsquigarrow \quad \exists X\colon (\varphi \land \chi) \lor (\psi \land \chi). \tag{4.6}$$

On the one hand, the rule allows to apply distribution of quantifiers over disjunction in the subsequent step (Equation (4.5)), pushing the quantifier deeper into the structure of the formula. On the other hand, we are performing more operations (three logical connectives instead of two). Therefore, we apply the rule only if the estimated cost of the operation (the size of the NFA) is smaller than a threshold. For estimating the cost of the operation, we used an approach based on machine learning with linear regression to learn how big an automaton for a given formula will be. With the use of the informed pre-processing, we managed to significantly improve the run-time of the underlying decision procedure, and even successfully solved a formula (obtained from an experimental translation of separation logic to WS*k*S) that had, to the best of our knowledge, not been solved by anyone before.

In addition to the logics WS1S/WS*k*S on finite words/trees, in [123], we also explored the logic S1S, which has the same syntax as WS1S (cf. Equation (4.1)), but a different semantics—existential quantifiers now range over *all* subsets of $\mathbb{N}$ and not only *finite*. The underlying formal model for deciding S1S are, therefore, Büchi (or, in general, ω-) automata. Although the complexities of problems for NFAs and BAs are the same (e.g., **PSPACE**-complete language inclusion) and so are the complexities for deciding WS1S and S1S (**TOWER**-complete), algorithms for working with BAs in practice scale much worse (e.g., for language complement, in the case of NFAs, a naive complementation algorithm usually suffices, but in the case of BAs, one needs to come up with quite elaborate algorithms to even start thinking about trying them in practice, cf. Chapter 2). It seems that as a consequence, with the exception of [29], there has been no attempt of implementing an S1S decision procedure in practice, despite the ubiquity of S1S (other than the logic itself, it can be used to encode other number-theoretical or temporal logics, e.g., QPTL [208]). In [123], we leveraged the recent advance in the algorithms for manipulating BAs and implemented a decision procedure for S1S, obtaining, in some cases, a much more compact representation than in [29].

4.2 Deciding Quantified Presburger Arithmetic

Another logic with an elegant automata-based decision procedure is the so-called Presburger arithmetic [192], i.e., the theory of natural numbers $\mathbb{N}$ with addition. In practice, the logic is often extended to reasoning over integers $\mathbb{Z}$, called *linear integer arithmetic* (LIA), using the following grammar:

$$\begin{aligned} \varphi_{atom} &::= \quad \vec{a}\cdot\vec{x} = c \mid \vec{a}\cdot\vec{x} \leq c \mid \vec{a}\cdot\vec{x} \equiv_m c \mid \bot \\ \varphi &::= \quad \varphi_{atom} \mid \neg\varphi \mid \varphi\wedge\varphi \mid \varphi\vee\varphi \mid \exists y(\varphi) \end{aligned}$$

where $\vec{a}$ is a (row) vector of n integer coefficients $(a_1,\ldots,a_n) \in \mathbb{Z}^n$, $\vec{x}$ is a (column) vector of n variables, $c \in \mathbb{Z}$ is a constant, $m \in \mathbb{Z}^+$ is a *modulus*, and y is a variable. The semantics of formulae is standard (in particular, $\vec{a}\cdot\vec{x} \equiv_m c$ holds iff there exists $z \in \mathbb{Z}$ s.t. $\vec{a}\cdot\vec{x} + z\cdot m = c$).

Current state-of-the-art SMT solvers, such as Z3 [177] or cvc5 [26], can routinely solve large quantifier-free (i.e., existential) LIA formulae, however, even a tiny formula with two quantifier alternations can be a show stopper for them. For instance, consider the formula

$$(\forall x, y \colon p \neq 7x + 11y) \land (\forall r(\neg \exists u, v \colon r = 7u + 11v) \Rightarrow r \leq p)$$

where p is the only free variable. The formula interpreted over natural numbers can be solved neither by Z3 nor by cvc5 in 1 minute (the model is $p = 59$). The formula encodes the so-called *Frobenius coin problem* [113], which asks "*What is the largest number p not expressible as a combination of numbers 7 and 11?*" Except being of theoretical interest, such formulae are obtained, e.g., during the analysis of Petri nets [77] or in some techniques for deciding the SMT theory of strings. Current SMT solvers also fail on many formulae coming directly from practice; cf. the `20190429-UltimateAutomizerSvcomp2019` benchmark in the LIA category of SMT-COMP (the international competition of SMT solvers) [211]. One reason for these inefficiencies is that SMT solvers deal with quantifiers by a syntactical approach called *quantifier instantiation* [106, 107, 176, 181, 196, 197, 199], without exploiting the structure of the underlying interpretation.

In our work, we focused on the automata-based decision procedure for LIA. The classical automata-based decision procedure for LIA is similar to the decision procedure for WS1S (cf. [40, 45, 229]). One can actually easily encode LIA into WS1S, in practice it is, however, more efficient, to use a specialized procedure, where the main difference is the treatment of atomic formulae (one also needs to handle treatment of negative numbers, but that is rather technical and will not be covered here). For instance, the DFA for the atomic formula $\vec{a} \cdot \vec{x} = c$ looks such that the set of states Q is a finite subset of $\mathbb{Z}$. For each integer $d \in Q$, it then holds that the language of the state labelled with d are all words that encode solutions of the formula $\vec{a} \cdot \vec{x} = d$. Therefore, $c \in \mathbb{Z}$ is the initial state of the DFA. The transition function given by the function *Post* for a symbol σ (representing the current bit value in the two's complement encoding of assignment to variables in $\vec{x}$) looks as follows:

$$Post(\vec{a} \cdot \vec{x} = c, \sigma) = \begin{cases} \vec{a} \cdot \vec{x} = \frac{1}{2}\kappa & \text{if } 2|\kappa \text{ for } \kappa = c - \vec{a} \cdot \sigma, \\ \bot & \text{otherwise.} \end{cases} \tag{4.7}$$

Here, instead of the state being just one integer, we use the whole formula.

When one takes the classical approach—which computes the DFA for the input formula inductively, starting from atomic formulae, and performing operations on DFAs corresponding to Boolean connectives—, it often blows up already when constructing the DFAs for atomic formulae. E.g., for some values of m the smallest DFA for a *congruence* $\vec{a} \cdot \vec{x} \equiv_m c$ can have m states. In practice, we encountered such DFAs for atomic formulae having over 300,000 states (in formulae obtained from ULTIMATE AUTOMIZER [125] during verification of computer programs).

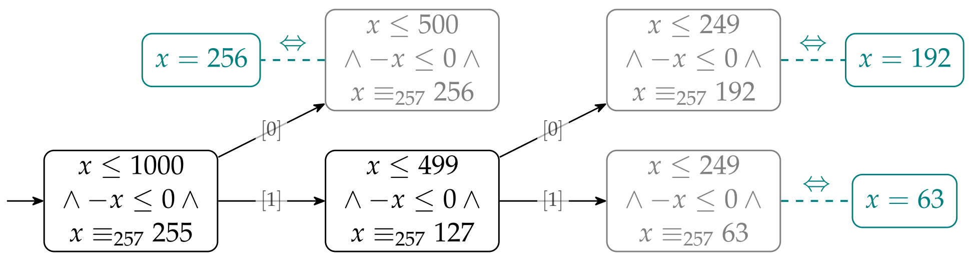

$x = 256$
$\Leftrightarrow$
$x \leq 500 \wedge -x \leq 0 \wedge x \equiv_{257} 256$
$x \leq 249 \wedge -x \leq 0 \wedge x \equiv_{257} 192$
$\Leftrightarrow$
$x = 192$
[0]
[0]
$x \leq 1000 \wedge -x \leq 0 \wedge x \equiv_{257} 255$
[1]
$x \leq 499 \wedge -x \leq 0 \wedge x \equiv_{257} 127$
[1]
$x \leq 249 \wedge -x \leq 0 \wedge x \equiv_{257} 63$
$\Leftrightarrow$
$x = 63$

Figure 4.3: Example of rewriting formulae/states in the NFA for the formula $x \leq 1000 \wedge -x \leq 0 \wedge x \equiv_{257} 255$.

Although the automata for atomic formulae can be large and the size can grow even larger when they are connected by Boolean connectives, we have observed that in many cases, the minimum DFAs for sub-formulae on a higher level in the AST of the input formula are, actually, quite small. Therefore, the performance of the decision procedure could get a significant boost if we could directly construct minimum (or at least small) automata for such formulae, which is what we try to do in [114].

In order to do this, we lift the idea of the lazy construction from [101, 118, 119] to LIA. In particular, we extend the definition of *Post* (cf. Equation (4.7)) from atomic formulae to arbitrary formulae. Now, the states of the constructed NFA are labelled by general LIA formulae: the language of the state equals encodings of all models of the LIA formula. This is the main difference from the procedures for WS1S/WS*k*S, where there was no direct correspondence of states and standard formulae (in [101, 118, 119] we needed to introduce the "automata terms", which contained derivatives). Having states corresponding to LIA formulae is powerful, since it allows us to perform *algebraic* simplifications of the formulae in states, in effect decreasing the size of the constructed NFA. For instance, consider the formula $x \leq 1000 \wedge -x \leq 0 \wedge x \equiv_{257} 255$. A part of the corresponding NFA can be seen in Figure 4.3. Some of the states (i.e., formulae) can, however, be significantly simplified, e.g., the formula in the state $x \leq 500 \wedge -x \leq 0 \wedge x \equiv_{257} 256$ can be simplified to $x = 256$. The new formula generates a much smaller state space (logarithmic compared to linear for the original formula).

Simplifying LIA formulae is hard in general (as hard as deciding their validity). We introduced several sub-optimal ways to simplify LIA formulae that work well in practice (the simplification is a trade-off between the complexity of simplifying a formula and the savings from the pruned state space). The first class of simplification rules is based on *pruning disjunctions* generated by unfolding existential quantifiers: given a state $\varphi_1 \vee \varphi_2$, if it holds that $\varphi_1 \Rightarrow \varphi_2$, then $\varphi_1 \vee \varphi_2$ can be replaced by just φ_1. Testing the entailment precisely is hard (in general as hard as deciding the original formula), so we use a stronger but cheaper relation of *subsumption*, which is defined inductively on the structure of formulae.

The second class of simplification rules is based on statically analyzing the formulae and inferring bounds on variables in order to perform *quantifier instantiation*, i.e., given a formula $\exists x(\varphi)$, obtaining an equivalent formula $\varphi[x/c]$ for the computed value of c (this can be, of course, done only sometimes). For instance, for the formula $\psi = \exists y(x - y \leq 1 \wedge y \leq -1 \wedge y \equiv_5 0)$, we can see that we want to substitute for y a number that is as big as possible (this follows from $x - y \leq 1$; the larger y is, the less it restricts x), but less than or equal to -1 (from $y \leq -1$), and, moreover, congruent to 0 modulo 5 (from $y \equiv_5 0$). This gives us the value $c = -5$, which allows us to, finally, rewrite ψ into the formula $x \leq -4$.

When compared to the formula simplifications carried out in classical SMT solvers, in our case, the simplifications are used on thousands of automata states, which dramatically amplifies their effect. We implemented the techniques in the solver Amaya and obtained a significant advantage over other state-of-the-art solvers on hard LIA formulae with quantifiers (cf. [114] for details). Also, we participated with Amaya in SMT-COMP'24 [211] and won the NIA (nonlinear integer arithmetic) logic sub-category of the Arith division in the 24 s Performance scoring scheme[1] and were second under three other scoring schemes.

In the future, we would like to extend the work to richer logics. For instance, extending LIA with predicates like "*x is a power of* 2" or "$y = x \cdot 2^k$" for a fixed k is trivial on the automata level but hard in other approaches. We would need to find new heuristics for the new predicates to work well with the rest of the framework. Moreover, it is an open question how to effectively and efficiently integrate automata-based theory solvers into state-of-the-art SMT solvers based on the Nelson-Oppen framework [178]. We would also like to explore the idea of extending the supported logic with uninterpreted functions, for which one could perhaps use some form of automata learning.

Contributed Papers[2]

[101] Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. "Lazy Automata Techniques for WS1S". In: *Proc. of TACAS'17*. Vol. 10205. LNCS. 2017, pp. 407–425.

[114] Peter Habermehl, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. "Algebraic Reasoning Meets Automata in Solving Linear Integer Arithmetic". In: *Proc. of CAV'24*. Vol. 14681. LNCS. Springer, 2024.

[1] How come our linear solver performed well in a category with nonlinear arithmetic? This is because in the SMT-LIB format, modulo congruence is considered as a non-linear operation, even though it can be rewritten using quantifiers into a linear equation.

[2] The work [119] (in bold) is attached to this thesis.

[117] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Ondřej Valeš, and Tomáš Vojnar. "Antiprenexing for WSkS: A Little Goes a Long Way". In: *Proc. of LPAR'20*. Vol. 73. EPiC Series in Computing. EasyChair, 2020, pp. 298–316.

[118] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Automata Terms in a Lazy WSkS Decision Procedure". In: *Proc. of CADE-27*. Vol. 11716. LNCS. Springer, 2019, pp. 300–318.

[119] **Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Automata Terms in a Lazy WSkS Decision Procedure". In: *J. Autom. Reason.* 65.7 (2021), pp. 971–999.**

[123] Vojtěch Havlena, Ondřej Lengál, and Barbora Šmahlíková. "Deciding S1S: Down the Rabbit Hole and Through the Looking Glass". In: *Proc. of NETYS'21*. Vol. 12754. LNCS. Springer, 2021, pp. 215–222.

Chapter 5

Theory of Strings

In computer science, manipulating *strings* of characters is ubiquitous. Most modern programming languages even have a special data type for storing strings. In formal reasoning over programs where string manipulation plays a role, one needs to have a support for string operations (string solving), in order to be able to detect security vulnerabilities such as SQL injection or cross-site scripting (XSS) in web applications [183–185]. String solving has also found its applications in, e.g., analysis of access user policies in Amazon Web Services [24, 163, 201] or smart contracts [17]. This led to the development of the SMT theory of strings [27] and decision procedures for this theory.

Over the years, many decision procedures and solvers for strings were developed. Many of them use automata, e.g., Stranger [235–237], Norn [5, 6], Ostrich [61–64, 166], Trau [2–4, 7], Sloth [131], Slog [226], Slent [225], Z3str3RE [33], Abc [22],or Bek [138]. Another class of solvers are those centered around word equations, for instance, cvc5 [26, 28, 164, 165, 182, 198, 200], Z3 [36, 177], S3 [215], Kepler$_{22}$ [156], StrSolve [139], Woorpje [84]. A different approach is using bit vectors, e.g., in Z3Str/2/3/4 [32, 174, 239, 240] or Hampi [148]. Some other techniques are based on using arrays (e.g., Pass [159]) or SAT solving (G-strings [19] and Gecode+S [206]). The approaches have different strengths and weaknesses with none of them being clearly the best in all cases.

In this chapter, we will give an overview of two of our approaches for string solving: first one based on regular model checking and then one based on the concept of the so-called *stabilization*.

5.1 String Solving using Regular Model Checking

An operation shared by the majority of approaches for dealing with *word equations* (i.e., constraints of the form $s = t$ where s and t are sequences of string variables and symbols) is the so-called *case-split rule* (implemented in different ways for different approaches). This rule represents the need to consider different possibilities of how a model looks like (e.g., alignment of variables on two sides of a word equation or whether the value of a variable is the empty word), with many of these possibilities yielding the same final outcome.

Consider, for instance, the so-called Nielsen transformation for solving word equations [179] and the equation $xz = yw$ with x, y, z, w being string variables. The transformation proceeds by first performing a case split based on the possible alignments of the variables x and y, the first symbols of the left and right-hand sides of the equation, respectively. More precisely, it reduces the satisfiability problem for $xz = yw$ into satisfiability of (at least) one of the following four cases (1) y is a prefix of x, (2) x is a prefix of y, (3) x is an empty string, and (4) y is an empty string. Note that these cases are not disjoint: for instance, the empty string is a prefix of every variable. For these four cases, the Nielsen transformation generates the following equations.

For the case (1), i.e., y is a prefix of x, all occurrences of x in $xz = yw$ are replaced with yx', where x' is a fresh word variable (we denote this case as $x \hookrightarrow yx'$), i.e., we obtain the equation $yx'z = yw$, which can be simplified to $x'z = w$. In fact, since the transformation $x \hookrightarrow yx'$ removes all occurrences of the variable x, we can just reuse the variable x and perform the transformation $x \hookrightarrow yx$ instead (and take this into account when constructing a model later).

Case (2) of the Nielsen transformation is just a symmetric counterpart of case (1) discussed above. For cases (3) and (4), x and y, respectively, are replaced by empty strings. Taking into account all four possible transformations of the equation $xz = yw$, we obtain the following four equations:

$$(1)\ xz = w, \qquad (2)\ z = yw, \qquad (3)\ z = yw, \quad \text{and} \qquad (4)\ xz = w.$$

(Note that the results for (1) and (4) coincide, as well as the results for (2) and (3).) If $xz = yw$ has a solution, then at least one of the above equations has a solution, too. The Nielsen transformation keeps applying the transformation rules on the obtained equations, building a proof tree and searching for a tautology of the form $\epsilon = \epsilon$.

In [71, 72], we came up with a compact encoding of the proof tree generated by Nielsen transformation (the technique should be also applicable to other approaches). We make use of the facts that (i) some of the obtained case splits are identical and (ii) many of the obtained case splits share a common sub-structure. For an example of (ii), consider the conjunction of word equations $xz = ab \wedge wabyx = awbzy$ (where a and b are symbols and x, y, z, w are string variables). After several rounds of applications of the rules above, one would obtain the following disjunction (representing the case split) of word equations: $wabyab = awby \vee wabya = awbby \vee waby = awbaby$ (see [71] for details). The equations could be represented by words over a two-track alphabet, where each track represents one side of the equation, as follows:

$$\binom{w}{a}\binom{a}{w}\binom{b}{b}\binom{y}{y}\binom{a}{\square}\binom{b}{\square}, \qquad \binom{w}{a}\binom{a}{w}\binom{b}{b}\binom{y}{b}\binom{a}{y}, \qquad \binom{w}{a}\binom{a}{w}\binom{b}{b}\binom{y}{a}\binom{\square}{b}\binom{\square}{y} \tag{5.1}$$

($\square$ is a *padding* symbol). The three words can be expressed using a single compact regular expression:

$$\binom{w}{a}\binom{a}{w}\binom{b}{b}\left[\binom{y}{y}\binom{a}{\square}\binom{b}{\square}+\binom{y}{b}\binom{a}{y}+\binom{y}{a}\binom{\square}{b}\binom{\square}{y}\right]. \tag{5.2}$$

In the work, we use NFAs to represent (i) the frontier of the proof tree and (ii) the set of all so-far generated nodes of the proof tree. Starting from an NFA $\mathcal{A}$ representing the input string constraint (the current frontier), we compute the *post-image* of $\mathcal{A}$ w.r.t. the transducer τ representing the Nielsen transformation rules (one can regard this as performing the *breadth-first search* through the proof tree). The result is also an NFA, which is used as the next frontier. The (finite length-preserving) transducer τ is constructed by translation from MSO(Strings) (monadic second-order logic over strings) formulae that describe the transformation rules. All of the obtained NFAs are united in an NFA $\mathcal{B}$ to keep track of all the traversed nodes of the proof tree. The previous is repeated until either we reach a frontier NFA whose language has a non-empty intersection with the language $\{\epsilon\}$ (plus padding, so in reality, we check for intersection with the language $\binom{\square}{\square}^*$) or the language of the frontier NFA is a subset of $\mathcal{B}$ (which means that we generated the whole proof tree without finding a terminal node). This approach can be formulated within the *regular model checking* (RMC) framework [1, 44].

Our approach is sound and is also complete for the so-called *quadratic* fragment of word equations, which are conjunctions of word equations where each variable has at most two occurrences (in the whole system). The completeness comes from finiteness of the proof tree (in a quadratic system, no transformation rule increases the number of variables in the system). We have also extended the approach past the quadratic fragment in several ways:

1. We extended the approach to the *cubic* fragment, i.e., conjunctions of word equations with at most three occurrences of every variable. Note that any conjunction of word equations can be transformed to the cubic form; one can do this by identifying a variable x with more than three occurrences, substituting two of its occurrences by a fresh variable y, and adding a word equation $x = y$. In the new system, y has exactly three occurrences and x has one less occurrence than it had previously. We can continue in the same manner until the system is cubic. While this extension preserves soundness, it does not preserve completeness—there is now no guarantee of termination.

2. Moreover, we also show how the technique can be applied on any Boolean combination of constraints. This is done by (i) rewriting negations to conjunctions and disjunctions using standard techniques from string solving [6], (ii) transforming the resulting formula into the conjunctive normal form (CNF) using distributive laws (one cannot use the Tseitin transformation [217] because it would re-introduce negations),

and (iii) encoding every clause as a set of words, which will become a part of the initial state.

3. One can also encode *regular constraints* into the framework. Regular constraints express that the assignment to a variable needs to fall into a given regular language (e.g., $x \in (ab)^*c$). The encoding is done by, intuitively, taking the NFA for the regular language and simulating a run through the NFA in case certain rules of the Nielsen transformation are taken. One can encode this in MSO(Strings) and extend the transducers in the standard way.

4. Finally, we also show how one can extend the approach to *length constraints*, for example $|x| < |y| + 2$. We extend the encoding with a part that talks about the lengths of the strings assigned to the variables, in a similar way as in automata-based decision procedures for Presburger arithmetic (cf. Section 4.2), and the transducers are, in turn, extended with manipulation of these lengths.

Prototype implementation of the resulting decision procedure showed that the approach is quite beneficial when used in a portfolio solver with other established solvers, such as Z3 [177] or cvc5 [26].

5.2 Stabilization-based String Solving

Many approaches for solving string constraints, including the one from the previous section, have issues when dealing with regular constraints (e.g., $x \in (a+b)^*c$). Often, regular constraints are handled only after, e.g., word equations are solved, and their handling is inefficient (e.g. the regular-constraint extension of the method in the previous section). This can cause exploring parts of the state space that the regular constraints exclude and also a considerable slow-down of the decision procedure.

In [38], we take an inverse approach—we work with regular constraints already from the beginning. The paper focuses on a fragment of string constraints that contains conjunctions of word equations and regular constraints (considering only conjunctions is usually sufficient, as this is the type of constraints that is passed to a theory solver within a Nelson-Oppen-based SMT solver [178]). For simplicity, consider a set of string variables $\mathbb{X}$ and a constraint of the form $x_1 \ldots x_m = x_{m+1} \ldots x_n \wedge \bigwedge_{x \in \mathbb{X}} L_x$ where $x_1, \ldots, x_n \in \mathbb{X}$ and L_x is a regular language for every $x \in \mathbb{X}$. Our approach is based on the following theorem:

Theorem 1 *The following are equivalent:*

1. *The constraint $x_1 \ldots x_m = x_{m+1} \ldots x_n \wedge \bigwedge_{x \in \mathbb{X}} L_x$ has a solution.*

2. *There exists an assignment $\nu\colon \mathbb{X} \to 2^{\Sigma^*}$ mapping every variable to a regular language such that (i) for every variable $x \in \mathbb{X}$, it holds that $\emptyset \subset \nu(x) \subseteq L_x$ and (ii) $\nu(x_1)\cdots\nu(x_m) = \nu(x_{m+1})\cdots\nu(x_n)$, i.e., the regular languages obtained by concatenating the assignments of languages to the variables on both sides of the equation coincide.*

Theorem 1 says that one can approach solving of the constraint by looking at the equation $L_{x_1}\cdots L_{x_m} = L_{x_{m+1}}\cdots L_{x_n}$—i.e., an equation obtained from the original word equation by substituting the variables with their languages—, and if the languages on the two sides of the equation do not equal, one can refine assignments of languages to the variables by removing strings from the languages that are forbidden by one of the sides of the equation. This can repeat until the equation holds, i.e., until *stabilization*.

For illustration, consider the string constraint $xy = yx \land x \in a(a+b)^* \land y \in (a+b)^+$ (we abuse notation and use regular expressions for describing regular languages). After substituting languages for the variables in the word equation $xy = yx$, we obtain the following:

$$\overbrace{a(a+b)^*}^{x}\overbrace{(a+b)^+}^{y} \neq \overbrace{(a+b)^+}^{y}\overbrace{a(a+b)^*}^{x}. \tag{5.3}$$

Since the equality of the regular languages does not hold, we refine the language assignment by constructing the intersection of the NFAs representing the two sides (while remembering which variables the corresponding parts belong to), inferring the following three possibilities to satisfy the equality:

$$N_1\colon \quad \overbrace{a(a+b)^*a(a+b)^*}^{x}\overbrace{(a+b)^+}^{y} = \overbrace{a(a+b)^*}^{y}\overbrace{a(a+b)^*(a+b)^+}^{x}$$

$$N_2\colon \quad \overbrace{a(a+b)^*}^{x}\overbrace{a(a+b)^*}^{y} = \overbrace{a(a+b)^*}^{y}\overbrace{a(a+b)^*}^{x}$$

$$N_3\colon \quad \overbrace{a(a+b)^*}^{x}\overbrace{(a+b)^+a(a+b)^*}^{y} = \overbrace{a(a+b)^*(a+b)^+}^{y}\overbrace{a(a+b)^*}^{x}$$

We call these three possibilities (N_1, N_2, and N_3) *noodles*. They denote a case split of possible ways how the variables on the two sides of an equation can match each other and are obtain from the NFA for the intersection of the two sides by taking the largest sub-NFAs with the same alignment of parts for the variables. For each of the noodles, we also need to synchronize all occurrences of the same variable. For instance, for N_1, on the left-hand side, x is assigned the language $a(a+b)^*a(a+b)^*$ and on the right-hand side, it is assigned $a(a+b)^*(a+b)^+$ (which can be simplified to $a(a+b)^+$). Their intersection is the language $a(a+b)^*a(a+b)^*$, which is assigned as the new language of x. For y, one would obtain the intersection of $(a+b)^+$ and $a(a+b)^*$, which is $a(a+b)^*$. We could then continue in N_1, again testing the equality of the new languages on the left-hand side and the right-hand side of the equation $xy = yx$. Instead, we could, however, take the noodle N_2, which assigns the language $a(a+b)^*$ to both x and y, and immediately makes the equation $L_x L_y = L_y L_x$ true. The proof of Theorem 1 shows how one can construct a string model if the regular language equality holds.

The technique is extended to systems of word equations, where the refinement of the variables' languages is done using all equations in some order. We have shown that for the *chain-free fragment* of word equations [7][1], the technique is complete, since the sequence of refinements terminates.

The technique was extended in [66] with the support of *length constraints*. The main idea of the extension is that when one obtains a stable assignment of languages to variables, instead of just terminating, one looks at the NFAs assigned to variables and obtains a quantifier-free LIA formula describing the lengths of the words from their languages (e.g., using a technique based on Parikh images [93, 187]), which can then be solved by an off-the-shelf LIA solver (which is an essential component of any practical SMT solver).

In order for this approach to be sound, the basic procedure, however, needs to be changed. The reason for this is that, in general, if we come to a stable variable assignment $\nu\colon \mathbb{X} \to 2^{\Sigma^*}$, while we know that there exists a model of the word equations compatible with ν, it does not need to hold that *any* assignment of strings from ν to the string variables is a model. Consider, e.g., the word equation $xy = yx$ and the language assignment $\nu = \{x \mapsto \{a, b\}, y \mapsto \{a, b\}\}$; while it holds that $\nu(x)\nu(y) = \nu(y)\nu(x)$, e.g., the string assignment $\{x \mapsto a, y \mapsto b\}$ is not a model (a model needs both variables to be assigned the same value).

The basic procedure is changed by adding the Align&Split rule (inspired by a similar technique originally presented in [6]), which, intuitively, performs an explicit alignment of variables on the two sides of an equation to each other (in which case they can be unified by replacing their occurrences by a single new variable) or splits one of them into two. This is used only for variables that have occurrences in some length constraint (the so-called length variables)—other variables can be treated as previously. The output assignment of languages to variables is then *monadic* (in the sense that any selection of strings from the language assignments represents a model) on the length variables.

In the same work [66], we also extended the framework with a support for string *disequations*—i.e., constraints of the form $t_1 \neq t_2$ with t_1 and t_2 being sequences of string variables—using a translation to word equations, as well as some other string predicates from the SMT-LIB format [27], such as `at`, `replace`, `indexof`, `contains`, `prefixof`, and `suffixof`. The procedure was implemented by replacing the string solver in Z3 [177] with our solver,

[1] whose definition is quite technical, but, intuitively, it prohibits cycles in a graph of dependencies between the sides of the equations

yielding a new SMT solver Z3-Noodler.

In [116], the solver was further extended with a support for string-integer conversions (SMT-LIB functions `from_code`, `from_int`, `to_code`, and `to_int`), which is performed by encoding the constraints into LIA formulae. This cannot be always performed (in general, we would need LIA extended with a 10^x function, which would yield an undecidable theory), but it is possible in many practical settings because usually, either (i) the language of strings to be converted into integers is finite or (ii) one can safely (while being sound for the SAT case) underapproximate the language to a finite language.

The solver Z3-Noodler with the techniques described in [38, 66, 67, 116] took part in the QF_Strings division of SMT-COMP'24 [211] (single query track) and won it under all scoring schemes, as well as the categories for the logics QF_S and QF_SLIA.

We are currently working on extending the framework with a better support for disequations and ¬`contains`/¬`prefixof`/¬`suffixof` predicates that, instead of translating the predicates (where it is possible) to word equations (which may potentially break the chain-freeness of the system), reduces the problem to configuration reachability in a version of counter automaton (or, in some cases, just a $\mathbb{Z}$-VASS). In addition, an extension that would allow to encode string transductions is also underway.

Contributed Papers[2]

[38] František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. "Word Equations in Synergy with Regular Constraints". In: *Proc. of FM'23*. Vol. 14000. LNCS. Springer, 2023, pp. 403–423.

[66] Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. "Solving String Constraints with Lengths by Stabilization". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 2112–2141.

[67] Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. "Z3-Noodler: An Automata-based String Solver". In: *Proc. of TACAS'24*. Vol. 14570. LNCS. Springer, 2024, pp. 24–33.

[71] **Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. "A symbolic algorithm for the case-split rule in solving word constraints with extensions". In: *J. Syst. Softw.* 201 (2023), p. 111673.**

[72] Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. "A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving". In: *APLAS'20*. Vol. 12470. LNCS. Springer, 2020, pp. 343–363.

[116] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. "Cooking String-Integer Conversions with Noodles". In: *Proc. of SAT'24*. Vol. 305. LIPIcs. Dagstuhl, 2024, 14:1–14:19.

[2]The work [71] (in bold) is attached to this thesis.

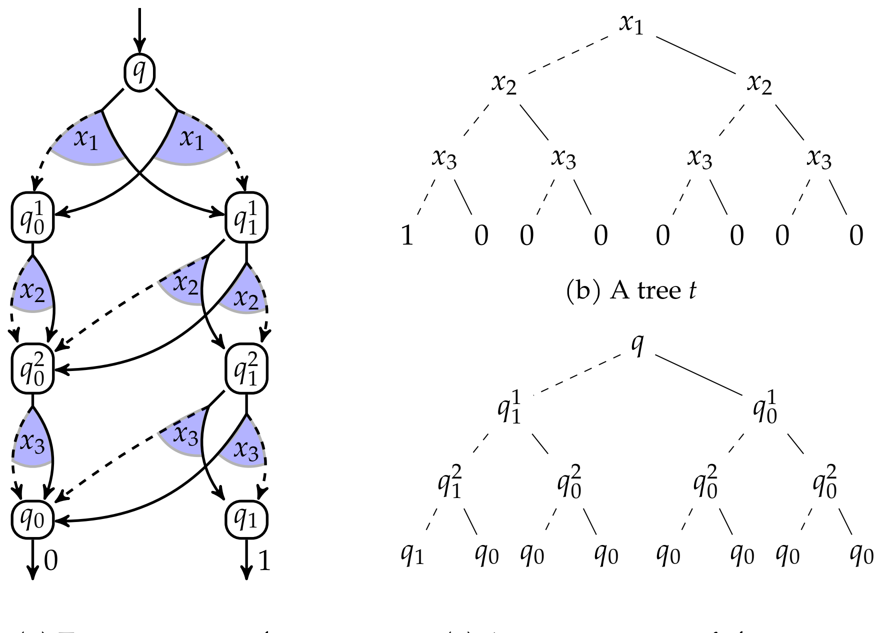
Chapter 6

Analysis of Quantum Circuits

The concept of *quantum computing* appeared around 1980 with the promise to solve many problems challenging for classical computers. Quantum algorithms for such problems started appearing later, such as Shor's factoring algorithm [207], a solution to the hidden subgroup problem by Ettinger *et al.* [96], Bernstein-Vazirani's algorithm [31], or Grover's search [111]. For a long time, no practical implementation of these algorithms has been available due to the missing hardware. Recent years have, however, seen the advent of quantum chips claiming to achieve *quantum supremacy* [21, 85], i.e., the ability to solve a problem that a state-of-the-art supercomputer would take thousands of years to solve. As it seems that quantum computers will occupy a prominent role in the future, systems and languages for their programming are in active development (e.g., [18, 110, 228]), and efficient quantum algorithms for solutions of real-world problems, such as machine learning [34, 78], optimization [172], or quantum chemistry [56], have started appearing.

The exponential size of the underlying computational space and the probabilistic nature makes it, however, extremely challenging to reason about quantum programs—both for human users and automated analysis tools. The existing automated analysis approaches are mostly unable to handle large-scale circuits [99, 100, 232–234], inflexible in checking user-specified properties [20, 54, 81, 97, 110, 180, 188, 203, 216, 222, 227, 241, 242], or imprecise and unable to catch bugs [189, 238]. In our work, we addressed these shortcomings and tried to develop scalable and flexible automated analysis tools for quantum circuits by exploiting ideas from automata theory and program analysis.

In [69], we introduced a novel framework for verification and bug hunting in quantum circuits. The framework is based on efficient representation of sets of quantum states by the use of finite tree automata [82]. The key idea is that a state of a quantum circuit with n qubits, which is a vector of 2^n complex numbers, can be seen as a *perfect binary tree* of the height n (i.e., a tree whose all branches have the same length n). Existing approaches represent one such a tree (i.e., a quantum state) compactly using variants of *decision*

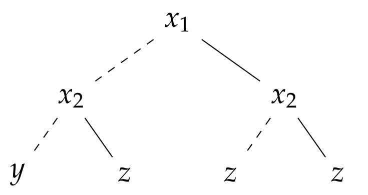


(a) Tree automaton $\mathcal{A}$ (c) An accepting run of $\mathcal{A}$ over t

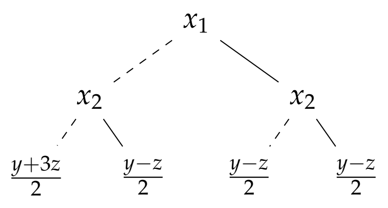
Figure 6.1: An example of (a) a tree automaton $\mathcal{A}$ encoding all computational basis states over 3 qubits (denoted as x_1, x_2, and x_3), (b) a tree t representing a quantum state, and (c) a run of $\mathcal{A}$ over t. Dashed parts of hyperedges go to the 0-child of the transitions, solid parts go to the 1-child of the transitions. Note that the TA generates the set of all perfect trees of the height 3 with one 1-leaf and all other leaves labelled by 0.

diagrams (DDs) [136, 137, 180, 209, 210, 216, 223, 224, 242]. For representing *sets of* trees (i.e., a predicate over quantum states) naively, one would need to have sets of the vectors or DDs. In common scenarios (e.g., when one considers all computational bases, i.e., 2^n quantum states, each representing a vector with 2^n complex values), this causes a blow-up in the size of the representation, not even mentioning the time complexity needed to execute operations over such a representation. In our framework, we encode *sets of quantum states* using *tree automata* (TAs). The use of TAs allows to compactly represent many sets of quantum states occurring in practice, e.g., the set of all computational basis states (a computational basis is a complex vector with exactly one occurrence of one and all other elements having the value zero); a TA representing all computational basis states and its run on an example tree-representation of a quantum state are depicted in Figure 6.1 (note that the size of the TA is linear to the number of qubits).

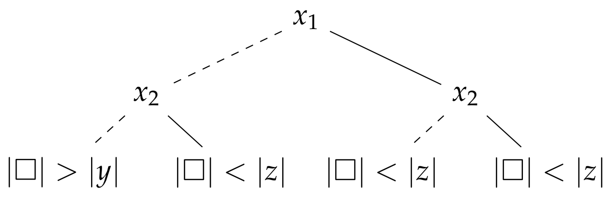
Our verification framework follows the Floyd-Hoare style of *deductive verification* based on the so-called *Hoare triples* $\{P\}\, C\, \{R\}$ with P and R being sets of quantum states and C being a quantum circuit. The Hoare triple de-



(a) A tree representing an input symbolic quantum state s



(b) An example of an output s' of a circuit with input s



(c) An example of a tree p representing a symbolic predicate over quantum states. When checking whether s' is represented by p, we substitute the $\square$ in the leaves with the term from the corresponding leaf of s' (e.g., for the leaf corresponding to $|00\rangle$, i.e., the left-most branch, we would obtain $\left|\frac{y+3z}{2}\right| > |y|$). We would then check whether $\varphi \Rightarrow \left|\frac{y+3z}{2}\right| > |y|$ and similarly for other branches.

Figure 6.2: An example of working with symbolic quantum states.

notes that if s is a quantum state from the set P and it is input to the circuit C, then the quantum state obtained from the output of the circuit belongs to the set R. In order to verify that a Hoare triple holds, we take a TA $\mathcal{A}_P$ representing P, run the circuit C on all trees from $\mathcal{A}_P$, obtaining a TA $\mathcal{A}'$, and check whether $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A}_R)$, where $\mathcal{A}_R$ is a TA representing R, using a standard TA language inclusion test. A crucial part of the procedure is defining *abstract transformers* that can execute quantum gates on all states represented by a TA efficiently. We have defined such transformers for a subset of quantum gates that is expressive enough to allow (approximate) universal quantum computation (the only standard gates not included are parameterized rotation gates, which can, however, be arbitrarily precisely approximated using the supported gates). The transformers for most of the gates (e.g., X, Z, or $CNOT$) perform a simple modification of the TA, which consists of redirecting some transitions, making copies of states, and potentially modifying the values of leaf transitions. For some gates (e.g., the *Hadamard* (H) gate), more complicated operations that require some synchronization in the trees, implemented by *tagging*, need to be performed. Our framework was able to prove correctness of a wide range of quantum circuits and we even managed to find a bug in the Qcec tool for quantum circuit equivalence checking [54].

We further extended the framework in a follow-up work [68] in two ways:

(i) Instead of only *concrete* quantum states, we also consider *symbolic* quantum states, whose representation uses symbolic variables and an accompanying global constraint formula φ. One such symbolic quantum state can represent many, potentially an infinite number of, concrete quantum states, based on the values of the symbolic variables (which need to satisfy φ). (ii) We introduced a *high-level specification language* that can be used for compactly describing correctness properties of quantum circuits, e.g., specifying that two consecutive H gates add up to identity, that the outputs of certain algorithms are quantum states with zero imaginary components of their amplitudes, or that Grover's search [111] outputs the correct result with a given probability and that one iteration of Grover's search performs amplification of the correct amplitude. Integrating the support for symbolic quantum states required redesigning the TA language inclusion test, since now the TAs contain symbolic variables and symbolic terms (obtained when performing gate operations with symbolic quantum states), cf. Figure 6.2.

The idea of working with symbolic variables was further developed in [65], now in the context of *simulation* of quantum circuits. The work developed the following two contributions to speeding up quantum simulation: (i) It introduces a precise representation of quantum states using *multi-terminal binary decision diagrams* (MTBDDs) [25, 105] and implementation of gate operations that is performed by special manipulation over the MTBDDs instead of performing generic *Apply* functions as done, e.g., in [216] in the context of BDD-based encoding. (ii) Mainly, it proposes an acceleration technique for quantum circuits that use loops with a fixed number of repetitions, such as various amplitude amplification algorithms [47] (including Grover's algorithm [111]), quantum counting [48], or period finding [149]. The acceleration works such that when a loop of the given structure is encountered during simulation, the simulation abstracts the quantum state entering the loop by introducing a symbolic variable for every concrete value in the quantum state. Then it executes the loop body, computing with algebraic terms instead of concrete values in the amplitudes. After the output is obtained, the algorithm checks whether the terms for all basis states with the same symbolic variable in the input are equivalent: if yes, we have computed a big-step semantics of the loop and can use it to iterate the given number of times to obtain the output quantum state; if not, we perform a refinement of the original abstraction and repeat the process. This technique was able to outperform other state-of-the-art simulators by several orders of magnitude.

As a future direction, we are currently exploring new formal models (modifications of TAs) that would allow some limited synchronization between left and right subtrees of tree nodes (this is required to compactly represent the effect of H (and some other) gates, which create a superposition). In addition, we are also working on extending the loop acceleration technique from simulation to symbolic verification and on an efficient support for simulating loops with measurements.

Contributed Papers[1]

[65] **Tian-Fu Chen, Yu-Fang Chen, Jie-Hong Roland Jiang, Ondřej Lengál, and Sára Jobranová. "Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization". In: *Proc. of ICCAD'24*. ACM, 2024.**

[68] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. "AutoQ: An Automata-Based Quantum Circuit Verifier". In: *Proc. of CAV'23*. Vol. 13966. LNCS. Springer, 2023, pp. 139–153.

[69] **Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. "An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1218–1243.**

[1]The works [65, 69] (in bold) are attached to this thesis.

Chapter 7

Conclusion

We have given an overview of the author's research in several areas related to finite-state automata, ranging from theoretical work (better upper bounds for complementation of classes of Büchi automata) to practical settings (e.g., approximative reduction of automata for the use in network intrusion detection or practical heuristics for deciding formulae of Presburger arithmetic). In order to be concise and keep the discussion on-target, we did not cover the results in other areas, such as efficient pattern matching using the so-called *counting (set) automata* (an extension of finite automata) [133, 218, 219], the research on *register automata* [76], computation of simulation over *symbolic automata* [134], verification of *parameterized concurrent systems* [157], *learning* of finite-state automata-based models of programs [75], verification of complex programs with *dynamic memory* [130], deciding *separation logic* using tree automata [91, 92], or specification of *concurrent programs* [74].

Appendix A contains one or more representative papers for each of the topics discussed in this thesis. Possible future research directions were given at the end of discussions of each of the topics, but they could be summarized as continuing in the endeavour to understand the structure of the addressed problems or their real-world instances and develop algorithms that exploit the structure and avoid the worst-case upper bounds, to, ultimately, obtain solutions that can be used in practice.

Bibliography

[1] P. A. Abdulla. "Regular model checking". In: *STTT* 14.2 (2012), pp. 109–118.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, D. Hu, W. Tsai, Z. Wu, and D. Yen. "Solving Not-Substring Constraint with Flat Abstraction". In: *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*. Ed. by H. Oh. Vol. 13008. Lecture Notes in Computer Science. Springer, 2021, pp. 305–320.

[3] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. "Flatten and conquer: a framework for efficient analysis of string constraints". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 602–617.

[4] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. "Trau: SMT solver for string constraints". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by N. S. Bjørner and A. Gurfinkel. IEEE, 2018, pp. 1–5.

[5] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. "Norn: An SMT Solver for String Constraints". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 462–469.

[6] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. "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*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 150–166.

[7] P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janku. "Chain-Free String Constraints". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Y. Chen, C. Cheng, and J. Esparza. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 277–293.

[8] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. "Composed Bisimulation for Tree Automata". In: *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*. Ed. by O. H. Ibarra and B. Ravikumar. Vol. 5148. Lecture Notes in Computer Science. Springer, 2008, pp. 212–222.

[9] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. “Computing Simulations over Tree Automata”. In: *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*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 93–108.

[10] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. “When Simulation Meets Antichains”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 158–174.

[11] P. A. Abdulla, Y. Chen, L. Holík, and T. Vojnar. “Mediating for reduction (on minimizing alternating Büchi automata)”. In: *Theor. Comput. Sci.* 552 (2014), pp. 26–43.

[12] P. A. Abdulla, L. Holík, L. Kaati, and T. Vojnar. “A Uniform (Bi-)Simulation-Based Framework for Reducing Tree Automata”. In: *Proceedings of the International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2008, Znojmo, Czech Republic, November 14-16, 2008*. Ed. by M. Ceska, Z. Kotásek, M. Kretínský, L. Matyska, and T. Vojnar. Vol. 251. Electronic Notes in Theoretical Computer Science. Elsevier, 2008, pp. 27–48.

[13] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d’Orso. “Regular Tree Model Checking”. In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Ed. by E. Brinksma and K. G. Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 555–568.

[14] O. Alexaj. “HyperLTL Model Checking”. Bachelor’s Thesis. Faculty of Information Technology, Brno University of Technology, 2024.

[15] J. D. Allred and U. Ultes-Nitsche. “A Simple and Optimal Complementation Algorithm for Büchi Automata”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. ACM, 2018, pp. 46–55.

[16] R. Almeida, L. Holík, and R. Mayr. “Reduction of Nondeterministic Tree Automata”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 717–735.

[17] L. Alt, M. Blicha, A. E. J. Hyvärinen, and N. Sharygina. “SolCMC: Solidity Compiler’s Model Checker”. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Ed. by S. Shoham and Y. Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 325–338.

[18] T. Altenkirch and J. Grattage. "A Functional Quantum Programming Language". In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 2005, pp. 249–258.

[19] R. Amadini, G. Gange, P. J. Stuckey, and G. Tack. "A Novel Approach to String Constraint Solving". In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by J. C. Beck. Vol. 10416. Lecture Notes in Computer Science. Springer, 2017, pp. 3–20.

[20] M. Amy. "Towards Large-scale Functional Verification of Universal Quantum Circuits". In: *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018*. Vol. 287. EPTCS. 2018, pp. 1–21.

[21] F. Arute et al. "Quantum supremacy using a programmable superconducting processor". en. In: *Nature* 574.7779 (Oct. 2019). Number: 7779 Publisher: Nature Publishing Group, pp. 505–510.

[22] A. Aydin, L. Bang, and T. Bultan. "Automata-Based Model Counting for String Constraints". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 255–272.

[23] T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Kretínský, D. Müller, D. Parker, and J. Strejcek. "The Hanoi Omega-Automata Format". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 479–486.

[24] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by N. S. Bjørner and A. Gurfinkel. IEEE, 2018, pp. 1–9.

[25] R. I. Bahar, E. A. Frohm, C. M. Gaona, et al. "Algebraic Decision Diagrams and Their Applications". In: *FMSD* 10.2/3 (1997), pp. 171–206.

[26] H. Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by D. Fisman and G. Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442.

[27] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB): Strings*. `https://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml`. 2024.

[28] C. W. Barrett, C. Tinelli, M. Deters, T. Liang, A. Reynolds, and N. Tsiskaridze. “Efficient solving of string constraints for security analysis”. In: *Proceedings of the Symposium and Bootcamp on the Science of Security, Pittsburgh, PA, USA, April 19-21, 2016*. Ed. by W. L. Scherlis and D. Brumley. ACM, 2016, pp. 4–6.

[29] S. Barth. “Deciding Monadic Second Order Logic over ω-Words by Specialized Finite Automata”. In: *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by E. Ábrahám and M. Huisman. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 245–259.

[30] D. A. Basin and N. Klarlund. “Hardware Verification using Monadic Second-Order Logic”. In: *Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings*. Ed. by P. Wolper. Vol. 939. Lecture Notes in Computer Science. Springer, 1995, pp. 31–41.

[31] E. Bernstein and U. V. Vazirani. “Quantum complexity theory”. In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*. ACM, 1993, pp. 11–20.

[32] M. Berzish, V. Ganesh, and Y. Zheng. “Z3str3: A string solver with theory-aware heuristics”. In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. Ed. by D. Stewart and G. Weissenbacher. IEEE, 2017, pp. 55–59.

[33] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh. “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*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 289–312.

[34] J. D. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. “Quantum machine learning”. In: *Nature* 549.7671 (2017), pp. 195–202.

[35] M. Biehl, N. Klarlund, and T. Rauhe. “Algorithms for Guided Tree Automata”. In: *Automata Implementation, First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, August 29-31, 1996, Revised Papers*. Ed. by D. R. Raymond, D. Wood, and S. Yu. Vol. 1260. Lecture Notes in Computer Science. Springer, 1996, pp. 6–25.

[36] N. S. Bjørner, N. Tillmann, and A. Voronkov. “Path Feasibility Analysis for String-Manipulating Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 307–321.

[37] F. Blahoudek, A. Duret-Lutz, and J. Strejcek. “Seminator 2 Can Complement Generalized Büchi Automata via Improved Semi-determinization”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by S. K. Lahiri and C. Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 15–27.

[38] F. Blahoudek, Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč. “Word Equations in Synergy with Regular Constraints”. In: *Proc. of FM’23*. Vol. 14000. LNCS. Springer, 2023, pp. 403–423.

[39] F. Blahoudek, M. Heizmann, S. Schewe, J. Strejček, and M. Tsai. “Complementing Semi-deterministic Büchi Automata”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 770–787.

[40] B. Boigelot, S. Jodogne, and P. Wolper. “An effective decision procedure for linear arithmetic over the integers and reals”. In: *ACM Trans. Comput. Log.* 6.3 (2005), pp. 614–633.

[41] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. “Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata”. In: *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*. Ed. by O. H. Ibarra and B. Ravikumar. Vol. 5148. Lecture Notes in Computer Science. Springer, 2008, pp. 57–67.

[42] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. “Abstract regular (tree) model checking”. In: *Int. J. Softw. Tools Technol. Transf.* 14.2 (2012), pp. 167–191.

[43] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. “Abstract Regular Tree Model Checking of Complex Dynamic Data Structures”. In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. Ed. by K. Yi. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 52–70.

[44] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. “Regular Model Checking”. In: *Computer Aided Verification*. Ed. by E. A. Emerson and A. P. Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 403–418.

[45] A. Boudet and H. Comon. “Diophantine Equations, Presburger Arithmetic and Finite Automata”. In: *Trees in Algebra and Programming - CAAP’96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*. Ed. by H. Kirchner. Vol. 1059. Lecture Notes in Computer Science. Springer, 1996, pp. 30–43.

[46] C. Brabrand, A. Møller, A. Sandholm, and M. I. Schwartzbach. “A Runtime System for Interactive Web Services”. In: *Comput. Networks* 31.11-16 (1999), pp. 1391–1401.

[47] G. Brassard, P. Høyer, M. Mosca, and A. Tapp. “Quantum amplitude amplification and estimation”. In: *Quantum computation and information* (*Washington, DC, 2000*). Vol. 305. Contemp. Math. Amer. Math. Soc., Providence, RI, 2002, pp. 53–74.

[48] G. Brassard, P. Høyer, and A. Tapp. “Quantum Counting”. In: *Automata, Languages and Programming, 25th International Colloquium, ICALP’98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. Vol. 1443. LNCS. Springer, 1998, pp. 820–831.

[49] S. Breuers, C. Löding, and J. Olschewski. "Improved Ramsey-Based Büchi Complementation". In: *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by L. Birkedal. Vol. 7213. Lecture Notes in Computer Science. Springer, 2012, pp. 150–164.

[50] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. "Logic and p-recognizable sets of integers". In: *Bulletin of the Belgian Mathematical Society - Simon Stevin* 1.2 (1994), pp. 191–238.

[51] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691.

[52] J. R. Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *Proc. of International Congress on Logic, Method, and Philosophy of Science 1962*. Stanford Univ. Press, Stanford, 1962.

[53] J. R. Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105`.

[54] L. Burgholzer and R. Wille. "Advanced equivalence checking for quantum circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.9 (2020), pp. 1810–1824.

[55] D. Bustan and O. Grumberg. "Simulation-based minimization". In: *ACM Trans. Comput. Log.* 4.2 (2003), pp. 181–206.

[56] Y. Cao et al. "Quantum Chemistry in the Age of Quantum Computing". In: *Chemical Reviews* 119.19 (2019). PMID: 31469277, pp. 10856–10915. eprint: `https://doi.org/10.1021/acs.chemrev.8b00803`.

[57] R. C. Carrasco and J. Oncina. "Learning Stochastic Regular Grammars by Means of a State Merging Method". In: *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*. Vol. 862. Lecture Notes in Computer Science. Springer, 1994, pp. 139–152.

[58] M. Češka, V. Havlena, L. Holík, J. Kořenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semrič, and T. Vojnar. "Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata". In: *Proc. of FCCM'19*. IEEE, 2019, pp. 109–117.

[59] M. Češka, V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection". In: *Proc. of TACAS'18*. Ed. by D. Beyer and M. Huisman. Vol. 10806. LNCS. Springer, 2018, pp. 155–175.

[60] M. Češka, V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Approximate reduction of finite automata for high-speed network intrusion detection". In: *Int. J. Softw. Tools Technol. Transf.* 22.5 (2020), pp. 523–539.

[61] T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. "What is decidable about string constraints with the ReplaceAll function". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 3:1–3:29.

[62] T. Chen, A. Flores-Lamas, M. Hague, Z. Han, D. Hu, S. Kan, A. W. Lin, P. Rümmer, and Z. Wu. "Solving string constraints with Regex-dependent functions through transducers with priorities and variables". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31.

[63] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu. "A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type". In: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. Ed. by D. V. Hung and O. Sokolsky. Vol. 12302. Lecture Notes in Computer Science. Springer, 2020, pp. 325–342.

[64] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. "Decision procedures for path feasibility of string-manipulating programs with complex operations". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 49:1–49:30.

[65] T. Chen, Y. Chen, J. R. Jiang, O. Lengál, and S. Jobranová. "Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization". In: *Proc. of ICCAD'24*. ACM, 2024.

[66] Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč. "Solving String Constraints with Lengths by Stabilization". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 2112–2141.

[67] Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč. "Z3-Noodler: An Automata-based String Solver". In: *Proc. of TACAS'24*. Vol. 14570. LNCS. Springer, 2024, pp. 24–33.

[68] Y. Chen, K. Chung, O. Lengál, J. Lin, and W. Tsai. "AutoQ: An Automata-Based Quantum Circuit Verifier". In: *Proc. of CAV'23*. Vol. 13966. LNCS. Springer, 2023, pp. 139–153.

[69] Y. Chen, K. Chung, O. Lengál, J. Lin, W. Tsai, and D. Yen. "An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1218–1243.

[70] Y. Chen, V. Havlena, and O. Lengál. "Simulations in Rank-Based Büchi Automata Complementation". In: *Proc. of APLAS'19*. Vol. 11893. LNCS. Springer, 2019, pp. 447–467.

[71] Y. Chen, V. Havlena, O. Lengál, and A. Turrini. "A symbolic algorithm for the case-split rule in solving word constraints with extensions". In: *J. Syst. Softw.* 201 (2023), p. 111673.

[72] Y. Chen, V. Havlena, O. Lengál, and A. Turrini. "A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving". In: *APLAS'20*. Vol. 12470. LNCS. Springer, 2020, pp. 343–363.

[73] Y. Chen, M. Heizmann, O. Lengál, Y. Li, M. Tsai, A. Turrini, and L. Zhang. "Advanced automata-based algorithms for program termination checking". In: *Proc. of PLDI'18*. ACM, 2018, pp. 135–150.

[74] Y. Chen, C. Hong, O. Lengál, S. Mu, N. Sinha, and B. Wang. "An Executable Sequential Specification for Spark Aggregation". In: *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*. Ed. by A. E. Abbadi and B. Garbinato. Vol. 10299. Lecture Notes in Computer Science. 2017, pp. 421–438.

[75] Y. Chen, C. Hsieh, O. Lengál, T. Lii, M. Tsai, B. Wang, and F. Wang. "PAC learning-based verification and model synthesis". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by L. K. Dillon, W. Visser, and L. A. Williams. ACM, 2016, pp. 714–724.

[76] Y. Chen, O. Lengál, T. Tan, and Z. Wu. "Register automata with linear arithmetic". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12.

[77] P. Chrzastowski-Wachtel and M. Raczunas. "Liveness of Weighted Circuits and the Diophantine Problem of Frobenius". In: *Fundamentals of Computation Theory, 9th International Symposium, FCT '93, Szeged, Hungary, August 23-27, 1993, Proceedings*. Ed. by Z. Ésik. Vol. 710. Lecture Notes in Computer Science. Springer, 1993, pp. 171–180.

[78] C. Ciliberto, M. Herbster, A. D. Ialongo, M. Pontil, A. Rocchetto, S. Severini, and L. Wossnig. "Quantum Machine Learning: A Classical Perspective". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 474.2209 (Jan. 2018).

[79] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Vol. 8414. LNCS. Springer, 2014, pp. 265–284.

[80] L. Clemente and R. Mayr. "Efficient reduction of nondeterministic automata with application to language inclusion testing". In: *Log. Methods Comput. Sci.* 15.1 (2019).

[81] B. Coecke and R. Duncan. "Interacting quantum observables: categorical algebra and diagrammatics". In: *New Journal of Physics* 13.4 (Apr. 2011), p. 043016.

[82] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. *Tree automata techniques and applications*. 2008.

[83] C. Courcoubetis and M. Yannakakis. "Verifying Temporal Properties of Finite-State Probabilistic Programs". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 338–345.

[84] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen. "On Solving Word Equations Using SAT". In: *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*. Ed. by E. Filiot, R. M. Jungers, and I. Potapov. Vol. 11674. Lecture Notes in Computer Science. Springer, 2019, pp. 93–106.

[85] M. DeCross et al. *The computational power of random quantum circuits in arbitrary geometries*. 2024. arXiv: `2406.02501 [quant-ph]`.

[86] L. Doyen and J. Raskin. "Antichain Algorithms for Finite Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 2–22.

[87] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. "Spot 2.0 - A Framework for LTL and ω-Automata Manipulation". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Ed. by C. Artho, A. Legay, and D. Peled. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 122–129.

[88] J. Elgaard, N. Klarlund, and A. Møller. "MONA 1.x: New Techniques for WS1S and WS2S". In: *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by A. J. Hu and M. Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 516–520.

[89] J. Elgaard, A. Møller, and M. I. Schwartzbach. "Compile-Time Debugging of C Programs Working on Trees". In: *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. Ed. by G. Smolka. Vol. 1782. Lecture Notes in Computer Science. Springer, 2000, pp. 119–134.

[90] E. A. Emerson and C. Lei. "Modalities for Model Checking: Branching Time Logic Strikes Back". In: *Sci. Comput. Program.* 8.3 (1987), pp. 275–306.

[91] C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. "Compositional entailment checking for a fragment of separation logic". In: *Formal Methods Syst. Des.* 51.3 (2017), pp. 575–607.

[92] C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. "SPEN: A Solver for Separation Logic". In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. Ed. by C. W. Barrett, M. D. Davies, and T. Kahsai. Vol. 10227. Lecture Notes in Computer Science. 2017, pp. 302–309.

[93] J. Esparza. "Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes". In: *Fundam. Informaticae* 31.1 (1997), pp. 13–25.

[94] J. Esparza and M. Blondin. *Automata Theory: An Algorithmic Approach*. The MIT Press, 2023.

[95] K. Etessami, T. Wilke, and R. A. Schuller. "Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata". In: *SIAM J. Comput.* 34.5 (2005), pp. 1159–1175.

[96] M. Ettinger, P. Høyer, and E. Knill. "The quantum query complexity of the hidden subgroup problem is polynomial". In: *Inf. Process. Lett.* 91.1 (2004), pp. 43–48.

[97] A. Fagan and R. Duncan. "Optimising Clifford Circuits with Quantomatic". In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), pp. 85–105.

[98] W. Feng, Y. Li, A. Turrini, M. Y. Vardi, and L. Zhang. "On the power of finite ambiguity in Büchi complementation". In: *Inf. Comput.* 292 (2023), p. 105032.

[99] Y. Feng, E. M. Hahn, A. Turrini, and S. Ying. "Model checking omega-regular properties for quantum Markov chains". In: *Proc. of CONCUR'17*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[100] Y. Feng, E. M. Hahn, A. Turrini, and L. Zhang. "QPMC: A Model Checker for Quantum Programs and Protocols". In: *International Symposium on Formal Methods*. Springer International Publishing, 2015, pp. 265–272.

[101] T. Fiedor, L. Holík, P. Janků, O. Lengál, and T. Vojnar. "Lazy Automata Techniques for WS1S". In: *Proc. of TACAS'17*. Vol. 10205. LNCS. 2017, pp. 407–425.

[102] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. "Nested Antichains for WS1S". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9035. LNCS. Springer, 2015, pp. 658–674.

[103] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. "Nested antichains for WS1S". In: *Acta Informatica* 56.3 (2019), pp. 205–228.

[104] E. Friedgut, O. Kupferman, and M. Y. Vardi. "Büchi Complementation Made Tighter". In: *Int. J. Found. Comput. Sci.* 17.4 (2006), pp. 851–868.

[105] M. Fujita, P. C. McGeer, and J. C. Yang. "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 149–169.

[106] Y. Ge, C. W. Barrett, and C. Tinelli. "Solving Quantified Verification Conditions Using Satisfiability Modulo Theories". In: *CADE-21*. Vol. 4603. LNCS. Springer, 2007, pp. 167–182.

[107] Y. Ge and L. M. de Moura. "Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories". In: *CAV'09*. Vol. 5643. LNCS. Springer, 2009, pp. 306–320.

[108] R. J. van Glabbeek and B. Ploeger. "Five Determinisation Algorithms". In: *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*. Ed. by O. H. Ibarra and B. Ravikumar. Vol. 5148. Lecture Notes in Computer Science. Springer, 2008, pp. 161–170.

[109] J. Glenn and W. I. Gasarch. "Implementing WS1S via Finite Automata". In: *Automata Implementation, First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, August 29-31, 1996, Revised Papers*. Ed. by D. R. Raymond, D. Wood, and S. Yu. Vol. 1260. Lecture Notes in Computer Science. Springer, 1996, pp. 50–63.

[110] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. "Quipper: a scalable quantum programming language". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* ACM, 2013, pp. 333–342.

[111] L. K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 1996, pp. 212–219.

[112] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. “On Complementing Nondeterministic Büchi Automata”. In: *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*. Ed. by D. Geist and E. Tronci. Vol. 2860. Lecture Notes in Computer Science. Springer, 2003, pp. 96–110.

[113] C. Haase. “A survival guide to Presburger arithmetic”. In: *ACM SIGLOG News* 5.3 (2018), pp. 67–82.

[114] P. Habermehl, V. Havlena, M. Hečko, L. Holík, and O. Lengál. “Algebraic Reasoning Meets Automata in Solving Linear Integer Arithmetic”. In: *Proc. of CAV'24*. Vol. 14681. LNCS. Springer, 2024.

[115] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar. “Forest Automata for Verification of Heap Manipulation”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 424–440.

[116] V. Havlena, L. Holík, O. Lengál, and J. Síč. “Cooking String-Integer Conversions with Noodles”. In: *Proc. of SAT'24*. Vol. 305. LIPIcs. Dagstuhl, 2024, 14:1–14:19.

[117] V. Havlena, L. Holík, O. Lengál, O. Valeš, and T. Vojnar. “Antiprenexing for WS*k*S: A Little Goes a Long Way”. In: *Proc. of LPAR'20*. Vol. 73. EPiC Series in Computing. EasyChair, 2020, pp. 298–316.

[118] V. Havlena, L. Holík, O. Lengál, and T. Vojnar. “Automata Terms in a Lazy WS*k*S Decision Procedure”. In: *Proc. of CADE-27*. Vol. 11716. LNCS. Springer, 2019, pp. 300–318.

[119] V. Havlena, L. Holík, O. Lengál, and T. Vojnar. “Automata Terms in a Lazy WS*k*S Decision Procedure”. In: *J. Autom. Reason.* 65.7 (2021), pp. 971–999.

[120] V. Havlena and O. Lengál. “Reducing (To) the Ranks: Efficient Rank-Based Büchi Automata Complementation”. In: *Proc. of CONCUR'21*. Vol. 203. LIPIcs. Dagstuhl, 2021, 2:1–2:19.

[121] V. Havlena, O. Lengál, Y. Li, B. Šmahlíková, and A. Turrini. “Modular Mix-and-Match Complementation of Büchi Automata”. In: *Proc. of TACAS'23*. Vol. 13993. LNCS. Springer, 2023, pp. 249–270.

[122] V. Havlena, O. Lengál, and B. Šmahlíková. “Complementing Büchi Automata with Ranker”. In: *Proc. of CAV'22*. Vol. 13372. LNCS. Springer, 2022, pp. 188–201.

[123] V. Havlena, O. Lengál, and B. Šmahlíková. “Deciding S1S: Down the Rabbit Hole and Through the Looking Glass”. In: *Proc. of NETYS'21*. Vol. 12754. LNCS. Springer, 2021, pp. 215–222.

[124] V. Havlena, O. Lengál, and B. Šmahlíková. "Sky Is Not the Limit - Tighter Rank Bounds for Elevator Automata in Büchi Automata Complementation". In: *Proc. of TACAS'22*. Vol. 13244. LNCS. Springer, 2022, pp. 118–136.

[125] M. Heizmann, J. Hoenicke, and A. Podelski. "Software Model Checking for People Who Love Automata". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by N. Sharygina and H. Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 36–52.

[126] M. Heizmann, J. Hoenicke, and A. Podelski. "Termination Analysis by Learning Terminating Programs". 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*. Vol. 8559. LNCS. Springer, 2014, pp. 797–813.

[127] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. "Mona: Monadic Second-Order Logic in Practice". In: *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*. Ed. by E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen. Vol. 1019. Lecture Notes in Computer Science. Springer, 1995, pp. 89–110.

[128] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. "Computing Simulations on Finite and Infinite Graphs". In: *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*. IEEE Computer Society, 1995, pp. 453–462.

[129] P. Hieronymi, D. Ma, R. Oei, L. Schaeffer, C. Schulz, and J. O. Shallit. "Decidability for Sturmian Words". In: *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*. Vol. 216. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 24:1–24:23.

[130] L. Holík, M. Hruska, O. Lengál, A. Rogalewicz, and T. Vojnar. "Counterexample Validation and Interpolation-Based Refinement for Forest Automata". In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. Ed. by A. Bouajjani and D. Monniaux. Vol. 10145. Lecture Notes in Computer Science. Springer, 2017, pp. 288–309.

[131] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 4:1–4:32.

[132] L. Holík, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. "Fully Automated Shape Analysis Based on Forest Automata". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by N. Sharygina and H. Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 740–755.

[133] L. Holík, O. Lengál, O. Saarikivi, L. Turonová, M. Veanes, and T. Vojnar. "Succinct Determinisation of Counting Automata via Sphere Construction". In: *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*. Ed. by A. W. Lin. Vol. 11893. Lecture Notes in Computer Science. Springer, 2019, pp. 468–489.

[134] L. Holík, O. Lengál, J. Síc, M. Veanes, and T. Vojnar. "Simulation Algorithms for Symbolic Automata". In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by S. K. Lahiri and C. Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 109–125.

[135] L. Holík, O. Lengál, J. Simácek, and T. Vojnar. "Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata". In: *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*. Ed. by T. Bultan and P. Hsiung. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 243–258.

[136] X. Hong, W. Huang, W. Chien, Y. Feng, M. Hsieh, S. Li, C. Yeh, and M. Ying. "Decision Diagrams for Symbolic Verification of Quantum Circuits". In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2023, pp. 970–977.

[137] X. Hong, X. Zhou, S. Li, Y. Feng, and M. Ying. "A Tensor Network based Decision Diagram for Representation of Quantum Circuits". In: *ACM Trans. Des. Autom. Electron. Syst.* 27.6 (June 2022).

[138] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. "Fast and Precise Sanitizer Analysis with BEK". In: *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[139] P. Hooimeijer and W. Weimer. "StrSolve: solving string constraints lazily". In: *Autom. Softw. Eng.* 19.4 (2012), pp. 531–559.

[140] L. Ilie, G. Navarro, and S. Yu. "On NFA Reductions". In: *Theory Is Forever, Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday*. Ed. by J. Karhumäki, H. A. Maurer, G. Paun, and G. Rozenberg. Vol. 3113. Lecture Notes in Computer Science. Springer, 2004, pp. 112–124.

[141] R. Iosif, A. Rogalewicz, and J. Simácek. "The Tree Width of Separation Logic with Recursive Definitions". In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Ed. by M. P. Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 21–38.

[142] J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. "Automatic Verification of Pointer Programs using Monadic Second-Order Logic". In: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*. Ed. by M. C. Chen, R. K. Cytron, and A. M. Berman. ACM, 1997, pp. 226–236.

[143] T. Jiang and B. Ravikumar. "Minimal NFA Problems are Hard". In: *SIAM J. Comput.* 22.6 (1993), pp. 1117–1141.

[144] G. Jirásková. "State complexity of some operations on binary regular languages". In: *Theor. Comput. Sci.* 330.2 (2005), pp. 287–298.

[145] D. Kähler and T. Wilke. "Complementation, Disambiguation, and Determinization of Büchi Automata Unified". In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*. Ed. by L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz. Vol. 5125. Lecture Notes in Computer Science. Springer, 2008, pp. 724–735.

[146] H. Karmarkar and S. Chakraborty. "On Minimal Odd Rankings for Büchi Complementation". In: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. Ed. by Z. Liu and A. P. Ravn. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 228–243.

[147] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. "MOSEL: A FLexible Toolset for Monadic Second-Order Logic". In: *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*. Ed. by E. Brinksma. Vol. 1217. Lecture Notes in Computer Science. Springer, 1997, pp. 183–202.

[148] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. "HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars". In: *ACM Trans. Softw. Eng. Methodol.* 21.4 (2012), 25:1–25:28.

[149] A. Y. Kitaev. "Quantum measurements and the Abelian Stabilizer Problem". In: *Electron. Colloquium Comput. Complex.* TR96-003 (1996). ECCC: TR96-003.

[150] N. Klarlund. "A Theory of Restrictions for Logics and Automata". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by N. Halbwachs and D. A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 406–417.

[151] N. Klarlund, A. Møller, and M. I. Schwartzbach. "MONA Implementation Secrets". In: *Int. J. Found. Comput. Sci.* 13.4 (2002), pp. 571–586.

[152] N. Klarlund, M. Nielsen, and K. Sunesen. "A Case Study in Verification Based on Trace Abstractions". In: *Formal Systems Specification, The RPC-Memory Specification Case Study (the book grow out of a Dagstuhl Seminar, September 1994)*. Ed. by M. Broy, S. Merz, and K. Spies. Vol. 1169. Lecture Notes in Computer Science. Springer, 1994, pp. 341–373.

[153] N. Klarlund, M. Nielsen, and K. Sunesen. "Automated Logical Verification Based on Trace Abstractions". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*. Ed. by J. E. Burns and Y. Moses. ACM, 1996, pp. 101–110.

[154] O. Kupferman and M. Y. Vardi. "Weak alternating automata are not that weak". In: *ACM Trans. Comput. Log.* 2.3 (2001), pp. 408–429.

[155] R. P. Kurshan. "Complementing Deterministic Büchi Automata in Polynomial Time". In: *J. Comput. Syst. Sci.* 35.1 (1987), pp. 59–71.

[156] Q. L. Le and M. He. "A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints". In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. Ed. by S. Ryu. Vol. 11275. Lecture Notes in Computer Science. Springer, 2018, pp. 350–372.

[157] O. Lengál, A. W. Lin, R. Majumdar, and P. Rümmer. "Fair Termination for Parameterized Probabilistic Concurrent Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by A. Legay and T. Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 499–517.

[158] O. Lengál, J. Simácek, and T. Vojnar. "VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by C. Flanagan and B. König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 79–94.

[159] G. Li and I. Ghosh. "PASS: String Solving with Parameterized Array and Interval Automaton". In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. Ed. by V. Bertacco and A. Legay. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 15–31.

[160] Y. Li, X. Sun, A. Turrini, Y. Chen, and J. Xu. "ROLL 1.0: ω-Regular Language Learning Library". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by T. Vojnar and L. Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 365–371.

[161] Y. Li, A. Turrini, W. Feng, M. Y. Vardi, and L. Zhang. "Divide-and-Conquer Determinization of Büchi Automata Based on SCC Decomposition". In: *Proc. of CAV'22*. Vol. 13372. LNCS. Springer, 2022, pp. 152–173.

[162] Y. Li, A. Turrini, L. Zhang, and S. Schewe. "Learning to Complement Büchi Automata". In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. Ed. by I. Dillig and J. Palsberg. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 313–335.

[163] Liana Hadarean. *String Solving at Amazon*. `https://mosca19.github.io/program/index.html`. Presented at MOSCA'19. 2019.

[164] T. Liang, A. Reynolds, C. Tinelli, C. W. Barrett, and M. Deters. "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions". 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*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 646–662.

[165] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. W. Barrett. "A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings". In: *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*. Ed. by C. Lutz and S. Ranise. Vol. 9322. Lecture Notes in Computer Science. Springer, 2015, pp. 135–150.

[166] A. W. Lin and P. Barceló. "String solving with word equations and transducers: towards a logic for analysing mutation XSS". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodík and R. Majumdar. ACM, 2016, pp. 123–136.

[167] C. Löding and A. Pirogov. "New Optimizations and Heuristics for Determinization of Büchi Automata". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Y. Chen, C. Cheng, and J. Esparza. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 317–333.

[168] D. Matousek, J. Korenek, and V. Pus. "High-speed regular expression matching with pipelined automata". In: *2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016*. Ed. by Y. Song, S. Wang, B. Nelson, J. Li, and Y. Peng. IEEE, 2016, pp. 93–100.

[169] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133.

[170] A. R. Meyer. "Weak Monadic Second Order Theory of Successor Is Not Elementary-recursive". In: *Logic Colloquium—Symposium on Logic Held at Boston, 1972–73*. Vol. 453. Lecture Notes in Mathematics. Springer, 1972, pp. 132–154.

[171] S. Miyano and T. Hayashi. "Alternating Finite Automata on omega-Words". In: *Theor. Comput. Sci.* 32 (1984), pp. 321–330.

[172] N. Moll et al. "Quantum optimization using variational algorithms on near-term quantum devices". In: *Quantum Science and Technology* 3.3 (June 2018), p. 030503.

[173] A. Møller and M. I. Schwartzbach. "The Pointer Assertion Logic Engine". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by M. Burke and M. L. Soffa. ACM, 2001, pp. 221–231.

[174] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh. "Z3str4: A Multi-armed String Solver". In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by M. Huisman, C. S. Pasareanu, and N. Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 389–406.

[175] F. Morawietz and T. Cornell. "The MSO Logic-Automaton Connection in Linguistics". In: *Logical Aspects of Computational Linguistics, Second International Conference, LACL '97, Nancy, France, September 22-24, 1997, Selected Papers*. Ed. by A. Lecomte, F. Lamarche, and G. Perrier. Vol. 1582. Lecture Notes in Computer Science. Springer, 1997, pp. 112–131.

[176] L. M. de Moura and N. Bjørner. "Efficient E-Matching for SMT Solvers". In: *CADE-21*. Vol. 4603. LNCS. Springer, 2007, pp. 183–198.

[177] L. M. de Moura and N. S. Bjørner. "Z3: An Efficient SMT Solver". In: *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*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[178] G. Nelson and D. C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Trans. Program. Lang. Syst.* 1.2 (1979), pp. 245–257.

[179] J. Nielsen. "Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden". In: *Mathematische Annalen* 78.1 (1917), pp. 385–397.

[180] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton, and R. Drechsler. "QMDDs: Efficient Quantum Function Representation and Manipulation". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35.1 (2016), pp. 86–99.

[181] A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli. "Syntax-Guided Quantifier Instantiation". In: *TACAS'21*. Vol. 12652. LNCS. Springer, 2021, pp. 145–163.

[182] A. Nötzli, A. Reynolds, H. Barbosa, C. W. Barrett, and C. Tinelli. "Even Faster Conflicts and Lazier Reductions for String Solvers". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by S. Shoham and Y. Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 205–226.

[183] OWASP. *Top 10*. `https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf`. 2013.

[184] OWASP. *Top 10*. `https://owasp.org/www-project-top-ten/2017/`. 2017.

[185] OWASP. *Top 10*. `https://owasp.org/Top10/`. 2021.

[186] R. Paige and R. E. Tarjan. "Three Partition Refinement Algorithms". In: *SIAM J. Comput.* 16.6 (1987), pp. 973–989.

[187] R. J. Parikh. "On Context-Free Languages". In: *J. ACM* 13.4 (Oct. 1966), pp. 570–581.

[188] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, E. W. Draeger, E. T. Holland, and R. Wisnieff. "Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral". In: *CoRR* abs/1710.05867 (2017).

[189] S. Perdrix. "Quantum entanglement analysis based on abstract interpretation". In: *International Static Analysis Symposium*. Springer. 2008, pp. 270–282.

[190] N. Piterman. "From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata". In: *Log. Methods Comput. Sci.* 3.3 (2007).

[191] A. Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[192] M. Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt". In: *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*. 1929, pp. 92–101.

[193] M. O. Rabin. "Decidability of second-order theories and automata on infinite trees." In: *Transactions of the American Mathematical Society* 141.0 (1969), pp. 1–35.

[194] R. R. Redziejowski. "An Improved Construction of Deterministic Omega-automaton Using Derivatives". In: *Fundam. Informaticae* 119.3-4 (2012), pp. 393–406.

[195] F. Renkin, P. Schlehuber-Caissier, A. Duret-Lutz, and A. Pommellet. "Effective Reductions of Mealy Machines". In: *Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*. Ed. by M. R. Mousavi and A. Philippou. Vol. 13273. Lecture Notes in Computer Science. Springer, 2022, pp. 114–130.

[196] A. Reynolds, H. Barbosa, and P. Fontaine. "Revisiting Enumerative Instantiation". In: *TACAS'18*. Vol. 10806. LNCS. Springer, 2018, pp. 112–131.

[197] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *CAV'15*. Vol. 9207. LNCS. Springer, 2015, pp. 198–216.

[198] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli. "Reductions for Strings and Regular Expressions Revisited". In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 225–235.

[199] A. Reynolds, C. Tinelli, and L. M. de Moura. "Finding conflicting instances of quantified formulas in SMT". In: *FMCAD'14*. IEEE, 2014, pp. 195–202.

[200] A. Reynolds, M. Woo, C. W. Barrett, D. Brumley, T. Liang, and C. Tinelli. "Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by R. Majumdar and V. Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 453–474.

[201] N. Rungta. "A Billion SMT Queries a Day (Invited Paper)". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Ed. by S. Shoham and Y. Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 3–18.

[202] S. Safra. "On the Complexity of omega-Automata". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 319–327.

[203] V. Samoladas. "Improved BDD Algorithms for the Simulation of Quantum Circuits". In: *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*. Vol. 5193. LNCS. Springer, 2008, pp. 720–731.

[204] S. Schewe. "Büchi Complementation Made Tight". In: *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*. Vol. 3. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009, pp. 661–672.

[205] S. Schmitz. "Complexity Hierarchies beyond Elementary". In: *ACM Trans. Comput. Theory* 8.1 (2016), 3:1–3:36.

[206] J. D. Scott, P. Flener, J. Pearson, and C. Schulte. "Design and Implementation of Bounded-Length Sequence Variables". In: *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*. Ed. by D. Salvagnin and M. Lombardi. Vol. 10335. Lecture Notes in Computer Science. Springer, 2017, pp. 51–67.

[207] P. W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134.

[208] A. P. Sistla, M. Y. Vardi, and P. Wolper. "The Complementation Problem for Büchi Automata with Applications to Temporal Logic". In: *Theor. Comput. Sci.* 49 (1987), pp. 217–237.

[209] M. Sistla, S. Chaudhuri, and T. W. Reps. "Symbolic Quantum Simulation with Quasimodo". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*. Vol. 13966. LNCS. Springer, 2023, pp. 213–225.

[210] M. A. Sistla, S. Chaudhuri, and T. Reps. "CFLOBDDs: Context-free-language ordered binary decision diagrams". In: *ACM Transactions on Programming Languages and Systems* (2023).

[211] *SMT-COMP'24*. `https://smt-comp.github.io/2024/`. 2024.

[212] J. W. Thatcher and J. B. Wright. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic". In: *Math. Syst. Theory* 2.1 (1968), pp. 57–81.

[213] The Snort Team. *Snort*. (`http://www.snort.org`).

[214] A. Tozawa and M. Hagiya. "XML Schema Containment Checking Based on Semi-implicit Techniques". In: *Implementation and Application of Automata, 8th International Conference, CIAA 2003, Santa Barbara, California, USA, July 16-18, 2003, Proceedings*. Ed. by O. H. Ibarra and Z. Dang. Vol. 2759. Lecture Notes in Computer Science. Springer, 2003, pp. 213–225.

[215] M. Trinh, D. Chu, and J. Jaffar. "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by G. Ahn, M. Yung, and N. Li. ACM, 2014, pp. 1232–1243.

[216] Y. Tsai, J. R. Jiang, and C. Jhang. "Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation". In: *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 439–444.

[217] G. S. Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[218] L. Turonová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar. "Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by K. R. B. Butler and K. Thomas. USENIX Association, 2022, pp. 4165–4182.

[219] L. Turonová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar. "Regex matching with counting-set automata". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 218:1–218:30.

[220] M. Y. Vardi and T. Wilke. "Automata: from logics to algorithms". In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. Ed. by J. Flum, E. Grädel, and T. Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 629–736.

[221] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 332–344.

[222] G. F. Viamontes, I. L. Markov, and J. P. Hayes. *Quantum Circuit Simulation*. Springer, 2009.

[223] L. Vinkhuijzen, T. Coopmans, D. Elkouss, V. Dunjko, and A. Laarman. "LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States". In: *Quantum* 7 (2023), p. 1108.

[224] L. Vinkhuijzen, T. Grurl, S. Hillmich, S. Brand, R. Wille, and A. Laarman. "Efficient Implementation of LIMDDs for Quantum Circuit Simulation". In: *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*. Vol. 13872. LNCS. Springer, 2023, pp. 3–21.

[225] H. Wang, S. Chen, F. Yu, and J. R. Jiang. "A symbolic model checking approach to the analysis of string and length constraints". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by M. Huchard, C. Kästner, and G. Fraser. ACM, 2018, pp. 623–633.

[226] H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang. "String Analysis via Automata Manipulation with Logic Circuit Representation". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 241–260.

[227] D. Wecker and K. M. Svore. "LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing". In: *CoRR* abs/1402.4467 (2014). arXiv: `1402.4467`.

[228] R. Wille, R. V. Meter, and Y. Naveh. "IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. IEEE, 2019, pp. 1234–1240.

[229] P. Wolper and B. Boigelot. “An Automata-Theoretic Approach to Presburger Arithmetic Constraints (Extended Abstract)”. In: *Static Analysis, Second International Symposium, SAS’95, Glasgow, UK, September 25-27, 1995, Proceedings*. Ed. by A. Mycroft. Vol. 983. Lecture Notes in Computer Science. Springer, 1995, pp. 21–32.

[230] M. D. Wulf, L. Doyen, T. A. Henzinger, and J. Raskin. “Antichains: A New Algorithm for Checking Universality of Finite Automata”. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by T. Ball and R. B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 17–30.

[231] Q. Yan. “Lower Bounds for Complementation of ω-Automata Via the Full Automata Technique”. In: *Log. Methods Comput. Sci.* 4.1 (2008).

[232] M. Ying. “Model Checking for Verification of Quantum Circuits”. In: *International Symposium on Formal Methods*. Springer. 2021, pp. 23–39.

[233] M. Ying and Y. Feng. *Model Checking Quantum Systems: Principles and Algorithms*. Cambridge University Press, 2021.

[234] M. Ying, Y. Li, N. Yu, and Y. Feng. “Model-checking linear-time properties of quantum systems”. In: *ACM Transactions on Computational Logic (TOCL)* 15.3 (2014), pp. 1–31.

[235] F. Yu, M. Alkhalaf, and T. Bultan. “Stranger: An Automata-Based String Analysis Tool for PHP”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 154–157.

[236] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. “Automata-based symbolic string analysis for vulnerability detection”. In: *Formal Methods Syst. Des.* 44.1 (2014), pp. 44–70.

[237] F. Yu, T. Bultan, and O. H. Ibarra. “Relational String Verification Using Multi-Track Automata”. In: *Int. J. Found. Comput. Sci.* 22.8 (2011), pp. 1909–1924.

[238] N. Yu and J. Palsberg. “Quantum abstract interpretation”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 542–558.

[239] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. “Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 235–254.

[240] Y. Zheng, X. Zhang, and V. Ganesh. “Z3-str: a Z3-based string solver for web application analysis”. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by B. Meyer, L. Baresi, and M. Mezini. ACM, 2013, pp. 114–124.

[241] A. Zulehner, S. Hillmich, and R. Wille. "How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing". In: *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*. ACM, 2019, pp. 1–7.

[242] A. Zulehner and R. Wille. "Advanced Simulation of Quantum Computations". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38.5 (2019), pp. 848–859.

Appendix A

Selected Papers

For each of the particular research directions discussed in this thesis, one to three representative papers are attached to the thesis. The following papers were selected:

[60] M. Češka, V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Approximate reduction of finite automata for high-speed network intrusion detection". In: *Int. J. Softw. Tools Technol. Transf.* 22.5 (2020), pp. 523–539.

[65] T. Chen, Y. Chen, J. R. Jiang, O. Lengál, and S. Jobranová. "Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization". In: *Proc. of ICCAD'24*. ACM, 2024.

[69] Y. Chen, K. Chung, O. Lengál, J. Lin, W. Tsai, and D. Yen. "An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1218–1243.

[71] Y. Chen, V. Havlena, O. Lengál, and A. Turrini. "A symbolic algorithm for the case-split rule in solving word constraints with extensions". In: *J. Syst. Softw.* 201 (2023), p. 111673.

[73] Y. Chen, M. Heizmann, O. Lengál, Y. Li, M. Tsai, A. Turrini, and L. Zhang. "Advanced automata-based algorithms for program termination checking". In: *Proc. of PLDI'18*. ACM, 2018, pp. 135–150.

[119] V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Automata Terms in a Lazy WS*k*S Decision Procedure". In: *J. Autom. Reason.* 65.7 (2021), pp. 971–999.

[121] V. Havlena, O. Lengál, Y. Li, B. Šmahlíková, and A. Turrini. "Modular Mix-and-Match Complementation of Büchi Automata". In: *Proc. of TACAS'23*. Vol. 13993. LNCS. Springer, 2023, pp. 249–270.

[124] V. Havlena, O. Lengál, and B. Šmahlíková. "Sky Is Not the Limit - Tighter Rank Bounds for Elevator Automata in Büchi Automata Complementation". In: *Proc. of TACAS'22*. Vol. 13244. LNCS. Springer, 2022, pp. 118–136.

Sky Is Not the Limit

Tighter Rank Bounds for Elevator Automata in Büchi Automata Complementation

Vojtěch Havlena, Ondřej Lengál✉, and Barbora Šmahlíková
ihavlena@fit.vut.cz, lengal@vut.cz, xsmahl00@vut.cz

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

Abstract. We propose several heuristics for mitigating one of the main causes of combinatorial explosion in rank-based complementation of Büchi automata (BAs): unnecessarily high bounds on the ranks of states. First, we identify *elevator automata*, which is a large class of BAs (generalizing semi-deterministic BAs), occurring often in practice, where ranks of states are bounded according to the structure of strongly connected components. The bounds for elevator automata also carry over to general BAs that contain elevator automata as a sub-structure. Second, we introduce two techniques for refining bounds on the ranks of BA states using data-flow analysis of the automaton. We implement out techniques as an extension of the tool Ranker for BA complementation and show that they indeed greatly prune the generated state space, obtaining significantly better results and outperforming other state-of-the-art tools on a large set of benchmarks.

1 Introduction

Büchi automata (BA) complementation has been a fundamental problem underlying many applications since it was introduced in 1962 by Büchi [8,17] as an essential part of a decision procedure for a fragment of the second-order arithmetic. BA complementation has been used as a crucial part of, e.g., termination analysis of programs [13,20,10] or decision procedures for various logics, such as S1S [8], the first-order logic of Sturmian words [33], or the temporal logics ETL and QPTL [38]. Moreover, BA complementation also underlies BA inclusion and equivalence testing, which are essential instruments in the BA toolbox. Optimal algorithms, whose output asymptotically matches the lower bound of $(0.76n)^n$ [43] (potentially modulo a polynomial factor), have been developed [37,1]. For a successful real-world use, asymptotic optimality is, however, not enough and these algorithms need to be equipped with a range of optimizations to make them behave better than the worst case on BAs occurring in practice.

In this paper, we focus on the so-called *rank-based* approach to complementation, introduced by Kupferman and Vardi [24], further improved with the help of Friedgut [14], and finally made optimal by Schewe [37]. The construction stores in a macrostate partial information about all runs of a BA $\mathcal{A}$ over some word α. In addition to tracking states that $\mathcal{A}$ can be in (which is sufficient, e.g., in the determinization of NFAs), a macrostate also stores a guess of the rank of each of the tracked states in the *run DAG* that captures all these runs. The guessed ranks impose restrictions on how the future of a state might look like (i.e., when $\mathcal{A}$ may accept). The number of macrostates in the complement

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 118–136, 2022.
https://doi.org/10.1007/978-3-030-99527-0_7

depends combinatorially on the maximum rank that occurs in the macrostates. The constructions in [24,14,37] provides only coarse bounds on the maximum ranks.

A way of decreasing the maximum rank has been suggested in [15] using a PSpace (and, therefore, not really practically applicable) algorithm (the problem of finding the optimal rank is PSpace-complete). In our previous paper [19], we have identified several basic optimizations of the construction that can be used to refine the *tight-rank upper bound* (TRUB) on the maximum ranks of states. In this paper, we push the applicability of rank-based techniques much further by introducing two novel lightweight techniques for refining the TRUB, thus significantly reducing the generated state space.

Firstly, we introduce a new class of the so-called *elevator automata*, which occur quite often in practice (e.g., as outputs of natural algorithms for translating LTL to BAs). Intuitively, an elevator automaton is a BA whose strongly connected components (SCCs) are all either inherently weak[1] or deterministic. Clearly, the class substantially generalizes the popular inherently weak [6] and semi-deterministic BAs [11,3,4]). The structure of elevator automata allows us to provide tighter estimates of the TRUBs, not only for elevator automata *per se*, but also for BAs where elevator automata occur as a sub-structure (which is even more common). Secondly, we propose a lightweight technique, inspired by data flow analysis, allowing to propagate rank restriction along the skeleton of the complemented automaton, obtaining even tighter TRUBs. We also extended the optimal rank-based algorithm to transition-based BAs (TBAs).

We implemented our optimizations within the Ranker tool [18] and evaluated our approach on thousands of hard automata from the literature (15 % of them were elevator automata that were not semi-deterministic, and many more contained an elevator sub-structure). Our techniques drastically reduce the generated state space; in many cases we even achieved exponential improvement compared to the optimal procedure of Schewe and our previous heuristics. The new version of Ranker gives a smaller complement in the majority of cases of hard automata than other state-of-the-art tools.

2 Preliminaries

Words, functions. We fix a finite nonempty alphabet Σ and the first infinite ordinal $\omega = \{0, 1, \ldots\}$. For $n \in \omega$, by $[n]$ we denote the set $\{0, \ldots, n\}$. For $i \in \omega$ we use $\llbracket i \rrbracket$ to denote the largest even number smaller of equal to i, e.g., $\llbracket 42 \rrbracket = \llbracket 43 \rrbracket = 42$. An (infinite) word α is represented as a function $\alpha \colon \omega \to \Sigma$ where the i-th symbol is denoted as α_i. We abuse notation and sometimes also represent α as an infinite sequence $\alpha = \alpha_0\alpha_1 \ldots$ We use Σ^ω to denote the set of all infinite words over Σ. For a (partial) function $f \colon X \to Y$ and a set $S \subseteq X$, we define $f(S) = \{f(x) \mid x \in S\}$. Moreover, for $x \in X$ and $y \in Y$, we use $f \triangleleft \{x \mapsto y\}$ to denote the function $(f \setminus \{x \mapsto f(x)\}) \cup \{x \mapsto y\}$.

Büchi automata. A (nondeterministic transition/state-based) *Büchi automaton* (BA) over Σ is a quadruple $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ where Q is a finite set of *states*, $\delta \colon Q \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is the sets of *initial* states, and $Q_F \subseteq Q$ and $\delta_F \subseteq \delta$ are the sets of *accepting states* and *accepting transitions* respectively. We sometimes treat δ as a set of transitions $p \xrightarrow{a} q$, for instance, we use $p \xrightarrow{a} q \in \delta$ to denote that $q \in \delta(p, a)$.

[1] An SCC is inherently weak if it either contains no accepting states or, on the other hand, all cycles of the SCC contain an accepting state.

Moreover, we extend δ to sets of states $P \subseteq Q$ as $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$, and to sets of symbols $\Gamma \subseteq \Sigma$ as $\delta(P, \Gamma) = \bigcup_{a \in \Gamma} \delta(P, a)$. We define the inverse transition function as $\delta^{-1} = \{p \xrightarrow{a} q \mid q \xrightarrow{a} p \in \delta\}$. The notation $\delta_{|_S}$ for $S \subseteq Q$ is used to denote the restriction of the transition function $\delta \cap (S \times \Sigma \times S)$. Moreover, for $q \in Q$, we use $\mathcal{A}[q]$ to denote the BA $(Q, \delta, \{q\}, Q_F \cup \delta_F)$.

A *run* of $\mathcal{A}$ from $q \in Q$ on an input word α is an infinite sequence $\rho \colon \omega \to Q$ that starts in q and respects δ, i.e., $\rho_0 = q$ and $\forall i \geq 0 \colon \rho_i \xrightarrow{\alpha_i} \rho_{i+1} \in \delta$. Let $\inf_Q(\rho)$ denote the states occurring in ρ infinitely often and $\inf_\delta(\rho)$ denote the transitions occurring in ρ infinitely often. The run ρ is called *accepting* iff $\inf_Q(\rho) \cap Q_F \neq \emptyset$ or $\inf_\delta(\rho) \cap \delta_F \neq \emptyset$.

A word α is accepted by $\mathcal{A}$ from a state $q \in Q$ if there is an accepting run ρ of $\mathcal{A}$ from q, i.e., $\rho_0 = q$. The set $\mathcal{L}_\mathcal{A}(q) = \{\alpha \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \alpha \text{ from } q\}$ is called the *language* of q (in $\mathcal{A}$). Given a set of states $R \subseteq Q$, we define the language of R as $\mathcal{L}_\mathcal{A}(R) = \bigcup_{q \in R} \mathcal{L}_\mathcal{A}(q)$ and the language of $\mathcal{A}$ as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(I)$. We say that a state $q \in Q$ is *useless* iff $\mathcal{L}_\mathcal{A}(q) = \emptyset$. If $\delta_F = \emptyset$, we call $\mathcal{A}$ *state-based* and if $Q_F = \emptyset$, we call $\mathcal{A}$ *transition-based*. In this paper, we fix a BA $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$.

3 Complementing Büchi automata

In this section, we describe a generalization of the rank-based complementation of state-based BAs presented by Schewe in [37] to our notion of transition/state-based BAs. Proofs can be found in [16].

3.1 Run DAGs

First, we recall the terminology from [37] (which is a minor modification of the one in [24]), which we use in the paper. Let the *run DAG* of $\mathcal{A}$ over a word α be a DAG (directed acyclic graph) $\mathcal{G}_\alpha = (V, E)$ containing vertices V and edges E such that

- $V \subseteq Q \times \omega$ s.t. $(q, i) \in V$ iff there is a run ρ of $\mathcal{A}$ from I over α with $\rho_i = q$,
- $E \subseteq V \times V$ s.t. $((q, i), (q', i')) \in E$ iff $i' = i + 1$ and $q' \in \delta(q, \alpha_i)$.

Given $\mathcal{G}_\alpha$ as above, we will write $(p, i) \in \mathcal{G}_\alpha$ to denote that $(p, i) \in V$. A vertex $(p, i) \in V$ is called *accepting* if p is an accepting state and an edge $((q, i), (q', i')) \in E$ is called *accepting* if $q \xrightarrow{\alpha_i} q'$ is an accepting transition. A vertex $v \in \mathcal{G}_\alpha$ is *finite* if the set of vertices reachable from v is finite, *infinite* if it is not finite, and *endangered* if it cannot reach an accepting vertex or an accepting edge.

We assign ranks to vertices of run DAGs as follows: Let $\mathcal{G}_\alpha^0 = \mathcal{G}_\alpha$ and $j = 0$. Repeat the following steps until the fixpoint or for at most $2n + 1$ steps, where $n = |Q|$.

- Set $\mathit{rank}_\alpha(v) \leftarrow j$ for all finite vertices v of $\mathcal{G}_\alpha^j$ and let $\mathcal{G}_\alpha^{j+1}$ be $\mathcal{G}_\alpha^j$ minus the vertices with the rank j.
- Set $\mathit{rank}_\alpha(v) \leftarrow j + 1$ for all endangered vertices v of $\mathcal{G}_\alpha^{j+1}$ and let $\mathcal{G}_\alpha^{j+2}$ be $\mathcal{G}_\alpha^{j+1}$ minus the vertices with the rank $j + 1$.
- Set $j \leftarrow j + 2$.

For all vertices v that have not been assigned a rank yet, we assign $\mathit{rank}_\alpha(v) \leftarrow \omega$.

We define the *rank of* α, denoted as $\mathit{rank}(\alpha)$, as $\max\{\mathit{rank}_\alpha(v) \mid v \in \mathcal{G}_\alpha\}$ and the *rank of* $\mathcal{A}$, denoted as $\mathit{rank}(\mathcal{A})$, as $\max\{\mathit{rank}(w) \mid w \in \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})\}$.

Lemma 1. *If $\alpha \notin \mathcal{L}(\mathcal{A})$, then $\mathit{rank}(\alpha) \leq 2|Q|$.*

3.2 Rank-Based Complementation

In this section, we describe a construction for complementing BAs developed in the work of Kupferman and Vardi [24]—later improved by Friedgut, Kupferman, and Vardi [14], and by Schewe [37]—extended to our definition of BAs with accepting states and transitions (see [19] for a step-by-step introduction). The construction is based on the notion of tight level rankings storing information about levels in run DAGs. For a BA $\mathcal{A}$ and $n = |Q|$, a *(level) ranking* is a function $f\colon Q \to [2n]$ such that $f(Q_F) \subseteq \{0, 2, \ldots, 2n\}$, i.e., f assigns even ranks to accepting states of $\mathcal{A}$. For two rankings f and f' we define $f \rightsquigarrow^a_S f'$ iff for each $q \in S$ and $q' \in \delta(q, a)$ we have $f'(q') \leq f(q)$ and for each $q'' \in \delta_F(q, a)$ it holds $f'(q'') \leq \llbracket f(q) \rrbracket$. The set of all rankings is denoted by $\mathcal{R}$. For a ranking f, the *rank* of f is defined as $\mathit{rank}(f) = \max\{f(q) \mid q \in Q\}$. We use $f \leq f'$ iff for every state $q \in Q$ we have $f(q) \leq f'(q)$ and we use $f < f'$ iff $f \leq f'$ and there is a state $q \in Q$ with $f(q) < f'(q)$. For a set of states $S \subseteq Q$, we call f to be *S-tight* if (i) it has an odd rank r, (ii) $f(S) \supseteq \{1, 3, \ldots, r\}$, and (iii) $f(Q \setminus S) = \{0\}$. A ranking is *tight* if it is Q-tight; we use $\mathcal{T}$ to denote the set of all tight rankings.

The original rank-based construction [24] uses macrostates of the form (S, O, f) to track all runs of $\mathcal{A}$ over α. The f-component contains guesses of the ranks of states in S (which is obtained by the classical subset construction) in the run DAG and the O-set is used to check whether all runs contain only a finite number of accepting states. Friedgut, Kupferman, and Vardi [14] improved the construction by having f consider only tight rankings. Schewe's construction [37] extends the macrostates to (S, O, f, i) with $i \in \omega$ representing a particular even rank such that O tracks states with rank i. At the cut-point (a macrostate with $O = \emptyset$) the value of i is changed to $i + 2$ modulo the rank of f. Macrostates in an accepting run hence iterate over all possible values of i. Formally, the complement of $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ is given as the (state-based) BA $\textsc{Schewe}(\mathcal{A}) = (Q', \delta', I', Q'_F \cup \emptyset)$, whose components are defined as follows:

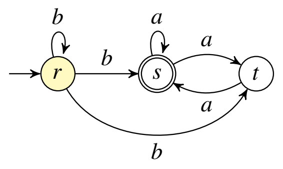
- $Q' = Q_1 \cup Q_2$ where
  - $Q_1 = 2^Q$ and
  - $Q_2 = \{(S, O, f, i) \in 2^Q \times 2^Q \times \mathcal{T} \times \{0, 2, \ldots, 2n - 2\} \mid f \text{ is } S\text{-tight}, O \subseteq S \cap f^{-1}(i)\}$,
- $I' = \{I\}$,
- $\delta' = \delta_1 \cup \delta_2 \cup \delta_3$ where
  - $\delta_1\colon Q_1 \times \Sigma \to 2^{Q_1}$ such that $\delta_1(S, a) = \{\delta(S, a)\}$,
  - $\delta_2\colon Q_1 \times \Sigma \to 2^{Q_2}$ such that $\delta_2(S, a) = \{(S', \emptyset, f, 0) \mid S' = \delta(S, a), f \text{ is } S'\text{-tight}\}$, and
  - $\delta_3\colon Q_2 \times \Sigma \to 2^{Q_2}$ such that $(S', O', f', i') \in \delta_3((S, O, f, i), a)$ iff
    - $S' = \delta(S, a)$,
    - $f \rightsquigarrow^a_S f'$,
    - $\mathit{rank}(f) = \mathit{rank}(f')$,
    - and
      - if $O = \emptyset$ then $i' = (i + 2) \mod (\mathit{rank}(f') + 1)$ and $O' = f'^{-1}(i')$, and
      - if $O \neq \emptyset$ then $i' = i$ and $O' = \delta(O, a) \cap f'^{-1}(i)$; and
- $Q'_F = \{\emptyset\} \cup ((2^Q \times \{\emptyset\} \times \mathcal{T} \times \omega) \cap Q_2)$.

We call the part of the automaton with states from Q_1 the *waiting* part (denoted as $\textsc{Waiting}$), and the part corresponding to Q_2 the *tight* part (denoted as $\textsc{Tight}$).

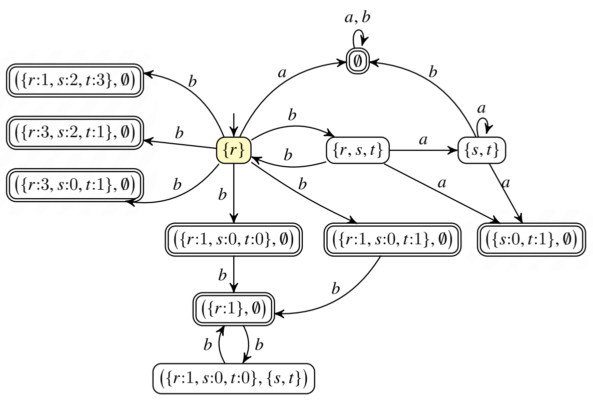
Theorem 2. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}(\textsc{Schewe}(\mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

The space complexity of Schewe's construction for BAs matches the theoretical lower bound $\mathcal{O}((0.76n)^n)$ given by Yan [43] modulo a quadratic factor $\mathcal{O}(n^2)$. Note that our extension to BAs with accepting transitions does not increase the space complexity of the construction.

Example 3. Consider the BA $\mathcal{A}$ over $\{a, b\}$ given in Fig. 1a. A part of $\textsc{Schewe}(\mathcal{A})$ is shown in Fig. 1b (we use $(\{s{:}0, t{:}1\}, \emptyset)$ to denote the macrostate $(\{s, t\}, \emptyset, \{s \mapsto 0, t \mapsto 1\}, 0))$. We omit the i-part of each macrostate since the corresponding values are 0 for all macrostates in the figure. Useless states are covered by grey stripes. The full automaton contains even more transitions from $\{r\}$ to useless macrostates of the form $(\{r{:}\cdot, s{:}\cdot, t{:}\cdot\}, \emptyset)$. □



(a) BA $\mathcal{A}$ over $\{a, b\}$



(b) A part of $\textsc{Schewe}(\mathcal{A})$

Fig. 1: Schewe's complementation

From the construction of $\textsc{Schewe}(\mathcal{A})$, we can see that the number of states is affected mainly by sizes of macrostates and by the maximum rank of $\mathcal{A}$. In particular, the upper bound on the number of states of the complement with the maximum rank r is given in the following lemma.

Lemma 4. *For a BA $\mathcal{A}$ with sufficiently many states n such that $rank(\mathcal{A}) = r$ the number of states of the complemented automaton is bounded by $2^n + \frac{(r+m)^n}{(r+m)!}$ where $m = \max\{0, 3 - \lceil \frac{r}{2} \rceil\}$.*

From Lemma 1 we have that the rank of $\mathcal{A}$ is bounded by $2|Q|$. Such a bound is often too coarse and hence $\textsc{Schewe}(\mathcal{A})$ may contain many redundant states. Decreasing the bound on the ranks is essential for a practical algorithm, but an optimal solution is PSPACE-complete [15]. The rest of this paper therefore proposes a framework of lightweight techniques for decreasing the maximum rank bound and, in this way, significantly reducing the size of the complemented BA.

3.3 Tight Rank Upper Bounds

Let $\alpha \notin \mathcal{L}(\mathcal{A})$. For $\ell \in \omega$, we define the ℓ-th *level* of $\mathcal{G}_\alpha$ as $level_\alpha(\ell) = \{q \mid (q, \ell) \in \mathcal{G}_\alpha\}$. Furthermore, we use f^α_ℓ to denote the ranking of level ℓ of $\mathcal{G}_\alpha$. Formally,

$$f^\alpha_\ell(q) = \begin{cases} rank_\alpha((q, \ell)) & \text{if } q \in level_\alpha(\ell), \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

We say that the ℓ-th level of $\mathcal{G}_\alpha$ is *tight* if for all $k \geq \ell$ it holds that (i) f^α_k is tight, and (ii) $rank(f^\alpha_k) = rank(f^\alpha_\ell)$. Let $\rho = S_0 S_1 \ldots S_{\ell-1}(S_\ell, O_\ell, f_\ell, i_\ell) \ldots$ be a run on a word

α in $\textsc{Schewe}(\mathcal{A})$. We say that ρ is a *super-tight run* [19] if $f_k = f_k^\alpha$ for each $k \geq \ell$. Finally, we say that a mapping $\mu\colon 2^Q \to \mathcal{R}$ is a *tight rank upper bound (TRUB) wrt α* iff

$$\exists \ell \in \omega\colon \mathit{level}_\alpha(\ell) \text{ is tight} \land (\forall k \geq \ell\colon \mu(\mathit{level}_\alpha(k)) \geq f_k^\alpha). \tag{2}$$

Informally, a TRUB is a ranking that gives a conservative (i.e., larger) estimate on the necessary ranks of states in a super-tight run. We say that μ is a TRUB iff μ is a TRUB wrt all $\alpha \notin \mathcal{L}(\mathcal{A})$. We abuse notation and use the term TRUB also for a mapping $\mu'\colon 2^Q \to \omega$ if the mapping $\mathit{inner}(\mu')$ is a TRUB where $\mathit{inner}(\mu')(S) = \{q \mapsto m \mid m = \mu'(S) \dot{-} 1 \text{ if } q \in Q_F \text{ else } m = \mu'(S)\}$ for all $S \in 2^Q$. ($\dot{-}$ is the *monus* operator, i.e., minus with negative results saturated to zero.) Note that the mappings $\mu_t = \{S \mapsto (2|S \setminus Q_F| \dot{-} 1)\}_{S \in 2^Q}$ and $\mathit{inner}(\mu_t)$ are trivial TRUBs.

The following lemma shows that we can remove from $\textsc{Schewe}(\mathcal{A})$ macrostates whose ranking is not covered by a TRUB (in particular, we show that the reduced automaton preserves super-tight runs).

Lemma 5. *Let μ be a TRUB and $\mathcal{B}$ be a BA obtained from $\textsc{Schewe}(\mathcal{A})$ by replacing all occurrences of Q_2 by $Q_2' = \{(S, O, f, i) \mid f \leq \mu(S)\}$. Then, $\mathcal{L}(\mathcal{B}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

4 Elevator Automata

In this section, we introduce *elevator automata*, which are BAs having a particular structure that can be exploited for complementation and semi-determinization; elevator automata can be complemented in $\mathcal{O}(16^n)$ (cf. Lemma 10) space instead of $2^{\mathcal{O}(n \log n)}$, which is the lower bound for unrestricted BAs, and semi-determinized in $\mathcal{O}(2^n)$ instead of $\mathcal{O}(4^n)$ (cf. [16]). The class of elevator automata is quite general: it can be seen as a substantial generalization of semi-deterministic BAs (SDBAs) [11,5]. Intuitively, an elevator automaton is a BA whose strongly connected components are all either deterministic or inherently weak.

Let $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$. $C \subseteq Q$ is a *strongly connected component* (SCC) of $\mathcal{A}$ if for any pair of states $q, q' \in C$ it holds that q is reachable from q' and q' is reachable from q. C is *maximal* (MSCC) if it is not a proper subset of another SCC. An MSCC C is *trivial* iff $|C| = 1$ and $\delta|_C = \emptyset$. The *condensation* of $\mathcal{A}$ is the DAG $\mathit{cond}(\mathcal{A}) = (\mathcal{M}, \mathcal{E})$ where $\mathcal{M}$ is the set of $\mathcal{A}$'s MSCCs and $\mathcal{E} = \{(C_1, C_2) \mid \exists q_1 \in C_1, \exists q_2 \in C_2, \exists a \in \Sigma\colon q_1 \xrightarrow{a} q_2 \in \delta\}$. An MSCC is *non-accepting* if it contains no accepting state and no accepting transition, i.e., $C \cap Q_F = \emptyset$ and $\delta|_C \cap \delta_F = \emptyset$. The *depth* of $(\mathcal{M}, \mathcal{E})$ is defined as the number of MSCCs on the longest path in $(\mathcal{M}, \mathcal{E})$.

We say that an SCC C is *inherently weak accepting* (IWA) iff *every cycle* in the transition diagram of $\mathcal{A}$ restricted to C contains an accepting state or an accepting transition. C is *inherently weak* if it is either non-accepting or IWA, and $\mathcal{A}$ is inherently weak if all of its MSCCs are inherently weak. $\mathcal{A}$ is *deterministic* iff $|I| \leq 1$ and $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. An SCC $C \subseteq Q$ is *deterministic* iff $(C, \delta|_C, \emptyset, \emptyset)$ is deterministic. $\mathcal{A}$ is a *semi-deterministic BA* (SDBA) if $\mathcal{A}[q]$ is deterministic for every $q \in Q_F \cup \{p \in Q \mid s \xrightarrow{a} p \in \delta_F, s \in Q, a \in \Sigma\}$, i.e., whenever a run in $\mathcal{A}$ reaches an accepting state or an accepting transition, it can only continue deterministically.

$\mathcal{A}$ is an *elevator (Büchi) automaton* iff for every MSCC C of $\mathcal{A}$ it holds that C is (i) deterministic, (ii) IWA, or (iii) non-accepting. In other words, a BA is an elevator automaton iff every nondeterministic SCC of $\mathcal{A}$ that contains an accepting state or transition is inherently weak. An example of an elevator automaton obtained from the LTL formula $\mathsf{GF}(a \vee \mathsf{GF}(b \vee \mathsf{GF}c))$ is shown in Fig. 2. The BA consists of three connected deterministic components. Note that the automaton is neither semi-deterministic nor unambiguous.

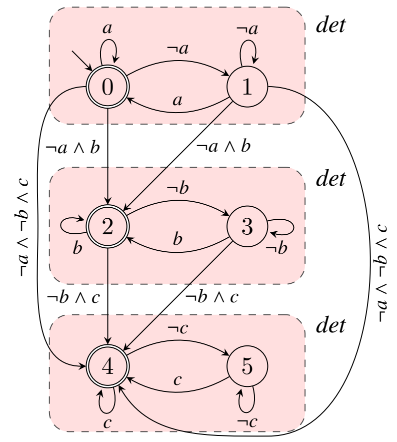


Fig. 2: The BA for LTL formula $\mathsf{GF}(a \vee \mathsf{GF}(b \vee \mathsf{GF}c))$ is elevator

The rank of an elevator automaton $\mathcal{A}$ does not depend on the number of states (as in general BAs), but only on the number of MSCCs and the depth of $cond(\mathcal{A})$. In the worst case, $\mathcal{A}$ consists of a chain of deterministic components, yielding the upper bound on the rank of elevator automata given in the following lemma.

Lemma 6. *Let $\mathcal{A}$ be an elevator automaton such that its condensation has the depth d. Then $rank(\mathcal{A}) \leq 2d$.*

4.1 Refined Ranks for Elevator Automata

Notice that the upper bound on ranks provided by Lemma 6 can still be too coarse. For instance, for an SDBA with three linearly ordered MSCCs such that the first two MSCCs are non-accepting and the last one is deterministic accepting, the lemma gives us an upper bound on the rank 6, while it is known that every SDBA has the rank at most 3 (cf. [5]). Another examples might be two deterministic non-trivial MSCCs connected by a path of trivial MSCCs, which can be assigned the same rank.

Instead of refining the definition of elevator automata into some quite complex list of constraints, we rather provide an algorithm that performs a traversal through $cond(\mathcal{A})$ and assigns each MSCC a label of the form $\boxed{type{:}rank}$ that contains (i) a type and (ii) a bound on the maximum rank of states in the component. The types of MSCCs that we consider are the following:

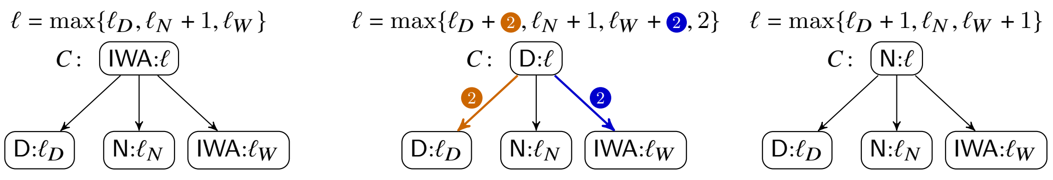
T: *trivial* components,
IWA: *inherently weak accepting* components,
D: *deterministic* (potentially accepting) components, and
N: *non-accepting* components.

Note that the type in an MSCC is not given *a priori* but is determined by the algorithm (this is because for deterministic non-accepting components, it is sometimes better to treated them as D and sometimes as N, depending on their neighbourhood). In the following, we assume that $\mathcal{A}$ is an elevator automaton without useless states and, moreover, all accepting conditions on states and transitions not inside non-trivial MSCCs are removed (any BA can be easily transformed into this form).

We start with terminal MSCCs C, i.e., MSCCs that cannot reach any other MSCC:

T1: If C is IWA, then we label it with $\boxed{\mathsf{IWA{:}0}}$.

T2: Else if C is deterministic accepting, we label it with $\boxed{\mathsf{D{:}2}}$.



ℓ = max{ℓ_D, ℓ_N + 1, ℓ_W}
C: IWA:ℓ
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
ℓ = max{ℓ_D + 2, ℓ_N + 1, ℓ_W + 2, 2}
C: D:ℓ
2
2
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
ℓ = max{ℓ_D + 1, ℓ_N, ℓ_W + 1}
C: N:ℓ
D:ℓ_D
N:ℓ_N
IWA:ℓ_W

(a) C is IWA (b) C is D (c) C is N

Fig. 3: Rules for assigning types and rank bounds to MSCCs. The symbols 2 and 2 are interpeted as 0 if all the corresponding edges from the components having rank ℓ_D and ℓ_W, respectively, are deterministic; otherwise they are interpreted as 2. Transitions between two components C_1 and C_2 are deterministic if the BA $(C, \delta|_C, \emptyset, \emptyset)$ is deterministic for $C = \delta(C_1, \Sigma) \cap (C_1 \cup C_2)$.

(Note that the previous two options are complete due to our requirements on the structure of $\mathcal{A}$.) When all terminal MSCCs are labelled, we proceed through $cond(\mathcal{A})$, inductively on its structure, and label non-terminal components C based on the rules defined below.

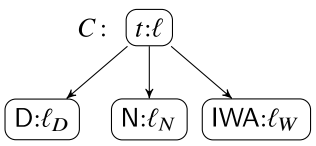


Fig. 4: Structure of elevator ranking rules

The rules are of the form that uses the structure depicted in Fig. 4, where children nodes denote already processed MSCCs. In particular, a child node of the form $\boxed{k{:}\ell_k}$ denotes an aggregate node of *all* siblings of the type k with ℓ_k being the maximum rank of these siblings. Moreover, we use $\mathrm{typemax}\{e_D, e_N, e_W\}$ to denote the type $j \in \{\mathsf{D}, \mathsf{N}, \mathsf{IWA}\}$ for which $e_j = \max\{e_D, e_N, e_W\}$ where e_i is an expression containing ℓ_i (if there are more such types, j is chosen arbitrarily). The rules for assigning a type t and a rank ℓ to C are the following:

I1: If C is trivial, we set $t = \mathrm{typemax}\{\ell_D, \ell_N, \ell_W\}$ and $\ell = \max\{\ell_D, \ell_N, \ell_W\}$.
I2: Else if C is IWA, we use the rule in Fig. 3a.
I3: Else if C is deterministic accepting, we use the rule in Fig. 3b.
I4: Else if C is deterministic and non-accepting, we try both rules from Figs. 3b and 3c and pick the rule that gives us a smaller rank.
I5: Else if C is nondeterministic and non-accepting, we use the rule in Fig. 3c.

Then, for every MSCC C of $\mathcal{A}$, we assign each of its states the rank of C. We use $\chi\colon Q \to \omega$ to denote the rank bounds computed by the procedure above.

Lemma 7. *χ is a TRUB.*

Using Lemma 5, we can now use χ to prune states during the construction of Schewe($\mathcal{A}$), as shown in the following example.

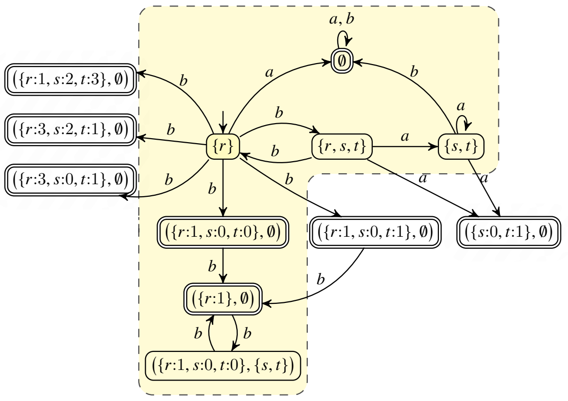


Fig. 5: A part of Schewe($\mathcal{A}$). The TRUB computed by elevator rules is used to prune states outside the yellow area.

Example 8. As an example, consider the BA $\mathcal{A}$ in Fig. 1a. The set of MSCCs with their types is given as

$\{\{r\}{:}\mathsf{N}, \{s,t\}{:}\mathsf{IWA}\}$ showing that BA $\mathcal{A}$ is an elevator. Using the rules **T1** and **I4** we get the TRUB $\chi = \{r{:}1, s{:}0, t{:}0\}$. The TRUB can be used to prune the generated states as shown in Fig. 5. □

4.2 Efficient Complementation of Elevator Automata

In Section 4.1 we proposed an algorithm for assigning ranks to MSCCs of an elevator automaton $\mathcal{A}$. The drawback of the algorithm is that the maximum obtained rank is not bounded by a constant but by the depth of the condensation of $\mathcal{A}$. We will, however, show that it is actually possible to change $\mathcal{A}$ by at most doubling the number of states and obtain an elevator BA with the rank at most 3.

Intuitively, the construction copies every non-trivial MSCC C with an accepting state or transition into a component $C^\bullet$, copies all transitions going into states in C to also go into the corresponding states in $C^\bullet$, and, finally, removes all accepting conditions from C. Formally, let $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ be a BA. For $C \subseteq Q$, we use $C^\bullet$ to denote a unique copy of C, i.e., $C^\bullet = \{q^\bullet \mid q \in C\}$ s.t. $C^\bullet \cap Q = \emptyset$. Let $\mathcal{M}$ be the set of MSCCs of $\mathcal{A}$. Then, the *deelevated* BA $\textsc{DeElev}(\mathcal{A}) = (Q', \delta', I', Q'_F \cup \delta'_F)$ is given as follows:

- $Q' = Q \cup Q^\bullet$,
- $\delta' : Q' \times \Sigma \to 2^{Q'}$ where for $q \in Q$
  - $\delta'(q,a) = \delta(q,a) \cup (\delta(q,a))^\bullet$ and
  - $\delta'(q^\bullet,a) = (\delta(q,a) \cap C)^\bullet$ for $q \in C \in \mathcal{M}$;
- $I' = I$, and
- $Q'_F = Q^\bullet_F$ and $\delta'_F = \{q^\bullet \xrightarrow{a} r^\bullet \mid q \xrightarrow{a} r \in \delta_F\} \cap \delta'$.

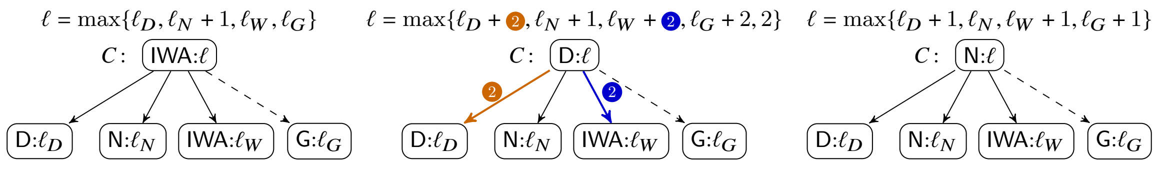
It is easy to see that the number of states of the deelevated automaton is bounded by $2|Q|$. Moreover, if $\mathcal{A}$ is elevator, so is $\textsc{DeElev}(\mathcal{A})$. The construction preserves the language of $\mathcal{A}$, as shown by the following lemma.

Lemma 9. *Let $\mathcal{A}$ be a BA. Then, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\textsc{DeElev}(\mathcal{A}))$.*

Moreover, for an elevator automaton $\mathcal{A}$, the structure of $\textsc{DeElev}(\mathcal{A})$ consists of (after trimming useless states) several non-accepting MSCCs with copied terminal deterministic or IWA MSCCs. Therefore, if we apply the algorithm from Section 4.1 on $\textsc{DeElev}(\mathcal{A})$, we get that its rank is bounded by 3, which gives the following upper bound for complementation of elevator automata.

Lemma 10. *Let $\mathcal{A}$ be an elevator automaton with suffficiently many states n. Then the language $\Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$ can be represented by a BA with at most $\mathcal{O}(16^n)$ states.*

The complementation through $\textsc{DeElev}(\mathcal{A})$ gives a better upper bound than the rank refinement from Section 4.1 applied directly on $\mathcal{A}$, however, based on our experience, complementation through $\textsc{DeElev}(\mathcal{A})$ behaves worse in many real-world instances. This poor behaviour is caused by the fact that the complement of $\textsc{DeElev}(\mathcal{A})$ can have a larger $\textsc{Waiting}$ and macrostates in $\textsc{Tight}$ can have larger S-components, which can yield more generated states (despite the rank bound 3). It seems that the most promising approach would to be a combination of the approaches, which we leave for future work.



$\ell = \max\{\ell_D, \ell_N + 1, \ell_W, \ell_G\}$
C: IWA:ℓ
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
G:ℓ_G
$\ell = \max\{\ell_D + 2, \ell_N + 1, \ell_W + 2, \ell_G + 2, 2\}$
C: D:ℓ
2
2
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
G:ℓ_G
$\ell = \max\{\ell_D + 1, \ell_N, \ell_W + 1, \ell_G + 1\}$
C: N:ℓ
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
G:ℓ_G

(a) C is IWA (b) C is D (c) C is N

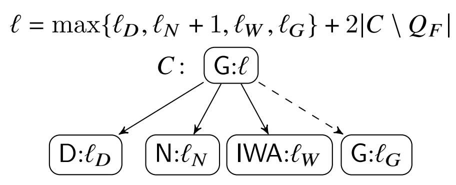
Fig. 6: Rules assigning types and rank bounds for non-elevator automata.

4.3 Refined Ranks for Non-Elevator Automata

The algorithm from Section 4.1 computing a TRUB for elevator automata can be extended to compute TRUBs even for general non-elevator automata (i.e., BAs with nondeterministic accepting components that are not inherently weak). To achieve this generalization, we extend the rules for assigning types and ranks to MSCCs of elevator automata from Section 4.1 to take into account general non-deterministic components. For this, we add into our collection of MSCC types *general* components (denoted as G). Further, we need to extend the rules for terminal components with the following rule:

T3: Otherwise, we label C with $\boxed{\mathsf{G}{:}2|C \setminus Q_F|}$.

Moreover, we adjust the rules for assigning a type t and a rank ℓ to C to the following (the rule **I1** is the same as for the case of elevator automata):



$\ell = \max\{\ell_D, \ell_N + 1, \ell_W, \ell_G\} + 2|C \setminus Q_F|$
C: G:ℓ
D:ℓ_D
N:ℓ_N
IWA:ℓ_W
G:ℓ_G

Fig. 7: C is G

I2–I5: *(We replace the corresponding rules for their counterparts including general components from Fig. 6).*
I6: Otherwise, we use the rule in Fig. 7.

Then, for every MSCC C of a BA $\mathcal{A}$, we assign each of its states the rank of C. Again, we use $\chi : Q \to \omega$ to denote the rank bounds computed by the adjusted procedure above.

Lemma 11. *χ is a TRUB.*

5 Rank Propagation

In the previous section, we proposed a way, how to obtain a TRUB for elevator automata (with generalization to general automata). In this section, we propose a way of using the structure of $\mathcal{A}$ to refine a TRUB using a propagation of values and thus reduce the size of TIGHT. Our approach uses *data flow analysis* [32] to reason on how ranks and rankings of macrostates of SCHEWE($\mathcal{A}$) can be decreased based on the ranks and rankings of the *local neighbourhood* of the macrostates. We, in particular, use a special case of *forward analysis* working on the *skeleton* of SCHEWE($\mathcal{A}$), which is defined as the BA $\mathcal{K}_{\mathcal{A}} = (2^Q, \delta', \emptyset, \emptyset)$ where $\delta' = \{R \xrightarrow{a} S \mid S = \delta(R, a)\}$ (note that we are only interested in the structure of $\mathcal{K}_{\mathcal{A}}$ and

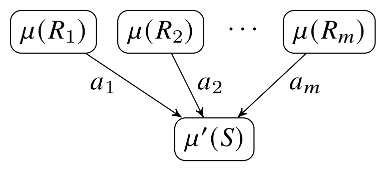


Fig. 8: Rank propagation flow

not its language; also notice the similarity of $\mathcal{K}_{\mathcal{A}}$ with WAITING). Our analysis refines a rank/ranking estimate $\mu(S)$ for a macrostate S of $\mathcal{K}_{\mathcal{A}}$ based on the estimates for its predecessors $R_1, \ldots, R_m$ (see Fig. 8). The new estimate is denoted as $\mu'(S)$.

More precisely, $\mu\colon 2^Q \to \mathbb{V}$ is a function giving each macrostate of $\mathcal{K}_{\mathcal{A}}$ a value from the domain $\mathbb{V}$. We will use the following two value domains: (i) $\mathbb{V} = \omega$, which is used for estimating *ranks* of macrostates (in the *outer macrostate analysis*) and (ii) $\mathbb{V} = \mathcal{R}$, which is used for estimating *rankings* within macrostates (in the *inner macrostate analysis*). For each of the analyses, we will give the *update function* $up\colon (2^Q \to \mathbb{V}) \times (2^Q)^{m+1} \to \mathbb{V}$, which defines how the value of $\mu(S)$ is updated based on the values of $\mu(R_1), \ldots, \mu(R_m)$. We then construct a system with the following equation for every $S \in 2^Q$:

$$\mu(S) = up(\mu, S, R_1, \ldots, R_m) \quad \text{where } \{R_1, \ldots, R_m\} = \delta'^{-1}(S, \Sigma). \tag{3}$$

We then solve the system of equations using standard algorithms for data flow analysis (see, e.g., [32, Chapter 2]) to obtain the fixpoint μ^*. Our analyses have the important property that if they start with μ_0 being a TRUB, then μ^* will also be a TRUB.

As the initial TRUB, we can use a trivial TRUB or any other TRUB (e.g., the output of elevator state analysis from Section 4).

5.1 Outer Macrostate Analysis

We start with the simpler analysis, which is the *outer macrostate analysis*, which only looks at sizes of macrostates. Recall that the rank r of every super-tight run in SCHEWE($\mathcal{A}$) does not change, i.e., a super tight run stays in WAITING as long as needed so that when it jumps to TIGHT, it takes the rank r and never needs to decrease it. We can use this fact to decrease the maximum rank of a macrostate S in $\mathcal{K}_{\mathcal{A}}$. In particular, let us consider all cycles going through S. For each of the cycles c, we can bound the maximum rank of a super-tight run going through c by $2m - 1$ where m is the smallest number of non-accepting states occurring in any macrostate on c (from the definition, the rank of a tight ranking does not depend on accepting states). Then we can infer that the maximum rank of any super-tight run going through S is bounded by the maximum rank of any of the cycles going through S (since S can never assume a higher rank in any super-tight run). Moreover, the rank of each cycle can also be estimated in a more precise way, e.g. using our elevator analysis.

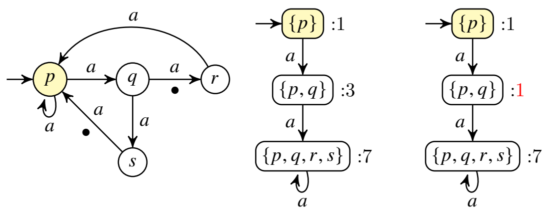
Since the number of cycles in $\mathcal{K}_{\mathcal{A}}$ can be large[2], instead of their enumeration, we employ data flow analysis with the value domain $\mathbb{V} = \omega$ (i.e, for every macrostate S of $\mathcal{K}_{\mathcal{A}}$, we remember a bound on the maximum rank of S) and the following update function:

$$up_{out}(\mu, S, R_1, \ldots, R_m) = \min\{\mu(S), \max\{\mu(R_1), \ldots, \mu(R_m)\}\}. \tag{4}$$

Intuitively, the new bound on the maximum rank of S is taken as the smaller of the previous bound $\mu(S)$ and the largest of the bounds of all predecessors of S, and the new value is propagated forward by the data flow analysis.

[2] $\mathcal{K}_{\mathcal{A}}$ can be exponentially larger than $\mathcal{A}$ and the number of cycles in $\mathcal{K}_{\mathcal{A}}$ can be exponential to the size of $\mathcal{K}_{\mathcal{A}}$, so the total number of cycles can be double-exponential.

Example 12. Consider the BA $\mathcal{A}_{ex}$ in Fig. 9a. When started from the initial TRUB $\mu_0 = \{\{p\} \mapsto 1, \{p,q\} \mapsto 3, \{p,q,r,s\} \mapsto 7\}$ (Fig. 9b), outer macrostate analysis decreases the maximum rank estimate for $\{p,q\}$ to 1, since $\min\{\mu_0(\{p,q\}, \max\{\mu_0(\{p\})\}\} = \min\{3,1\} = 1$. The estimate for $\{p,q,r,s\}$ is not affected, because $\min\{7, \max\{1,7\}\} = 7$ (Fig. 9c). □



(a) $\mathcal{A}_{ex}$ (b) μ_0 (c) μ^*_{out}

Fig. 9: Example of outer macrostate analysis. (a) $\mathcal{A}_{ex}$ (● denotes accepting transitions). The initial TRUB μ_0 in (b) is refined to μ^*_{out} in (c).

Lemma 13. *If μ is a TRUB, then $\mu \lhd \{S \mapsto up_{out}(\mu, S, R_1, \ldots, R_m)\}$ is a TRUB.*

Corollary 14. *When started with a TRUB μ_0, the outer macrostate analysis terminates and returns a TRUB μ^*_{out}.*

5.2 Inner Macrostate Analysis

Our second analysis, called *inner macrostate analysis*, looks deeper into super-tight runs in Schewe($\mathcal{A}$). In particular, compared with the outer macrostate analysis from the previous section—which only looks at the *ranks*, i.e., the bounds on the numbers in the rankings—, inner macrostate analysis looks at how the *rankings* assign concrete values to the *states* of $\mathcal{A}$ *inside the macrostates*.

Inner macrostate analysis is based on the following. Let ρ be a super-tight run of Schewe($\mathcal{A}$) on $\alpha \notin \mathcal{L}(\mathcal{A})$ and (S, O, f, i) be a macrostate from Tight. Because ρ is super-tight, we know that the rank $f(q)$ of a state $q \in S$ is bounded by the ranks of the predecessors of q. This holds because in super-tight runs, the ranks are only *as high as necessary*; if the rank of q were higher than the ranks of its predecessors, this would mean that we may wait in Waiting longer and only jump to q with a lower rank later.

Let us introduce some necessary notation. Let $f, f' \in \mathcal{R}$ be rankings (i.e., $f, f' \colon Q \to \omega$). We use $f \sqcup f'$ to denote the ranking $\{q \mapsto \max\{f(q), f'(q)\} \mid q \in Q\}$, and $f \sqcap f'$ to denote the ranking $\{q \mapsto \min\{f(q), f'(q)\} \mid q \in Q\}$. Moreover, we define *max-succ-rank*$^a_S(f) = \max_{\leq}\{f' \in \mathcal{R} \mid f \xrightarrow{a}_S f'\}$ and a function $dec \colon \mathcal{R} \to \mathcal{R}$ such that $dec(\theta)$ is the ranking θ' for which

$$\theta'(q) = \begin{cases} \theta(q) \dot{-} 1 & \text{if } \theta(q) = rank(\theta) \text{ and } q \notin Q_F, \\ \llbracket \theta(q) \dot{-} 1 \rrbracket & \text{if } \theta(q) = rank(\theta) \text{ and } q \in Q_F, \\ \theta(q) & \text{otherwise.} \end{cases} \qquad (5)$$

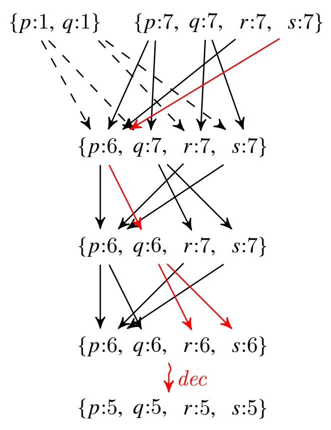
Intuitively, *max-succ-rank*$^a_S(f)$ is the (pointwise) maximum ranking that can be reached from macrostate S with ranking f over a (it is easy to see that there is a unique such maximum ranking) and $dec(\theta)$ decreases the maximum ranks in a ranking θ by one (or by two for even maximum ranks and accepting states).

The analysis uses the value domain $\mathbb{V} = \mathcal{R}$ (i.e., each macrostate of $\mathcal{K}_{\mathcal{A}}$ is assigned a ranking giving an upper bound on the rank of each state in the macrostate) and the update function up_{in} given in the right-hand side of the page. Intuitively, up_{in}

updates $\mu(q)$ for every $q \in S$ to hold the maximum rank compatible with the ranks of its predecessors. We note line Line 6, which makes use of the fact that we can only consider tight rankings (whose rank is odd), so we can decrease the estimate using the function *dec* defined above.

```
up_in(μ, S, R_1, ..., R_m):
    foreach $1 \le i \le m$ and $a \in \Sigma$ do
        if $\delta(R_i, a) = S$ then
            $g_i^a \leftarrow$ max-succ-rank$^a_{R_i}(\mu(R_i))$
    $\theta \leftarrow \mu(S) \sqcap \bigsqcup\{g_i^a \mid g_i^a \text{ is defined}\}$;
    if $rank(\theta)$ is even then $\theta \leftarrow dec(\theta)$;
    return $\theta$;
```

Example 15. Let us continue in Section 5.1 and perform inner macrostate analysis starting with the TRUB $\{\{p{:}1\}, \{p{:}1, q{:}1\}, \{p{:}7, q{:}7, r{:}7, s{:}7\}\}$ obtained from μ^*_{out}. We show three iterations of the algorithm for $\{p, q, r, s\}$ in the right-hand side (we do not show $\{p, q\}$ except the first iteration since it does not affect intermediate steps). We can notice that in the three iterations, we could decrease the maximum rank estimate to $\{p{:}6, q{:}6, r{:}6, s{:}6\}$ due to the accepting transitions from r and s. In the last of the three iterations, when all states have the even rank 6, the condition on Line 6 would become true and the rank of all states would be decremented to 5 using *dec*. Then, again, the accepting transitions from r and s would decrease the rank of p to 4, which would be propagated to q and so on. Eventually, we would arrive to the TRUB $\{p{:}1, q{:}1, r{:}1, s{:}1\}$, which could not be decreased any more, since $\{p{:}1, q{:}1\}$ forces the ranks of r and s to stay at 1. □



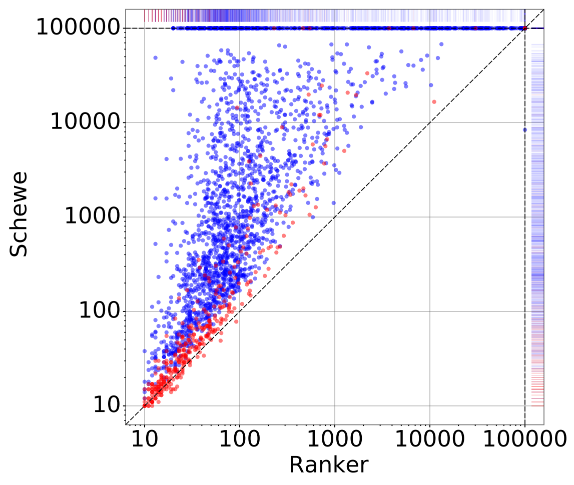
Lemma 16. *If μ is a TRUB, then $\mu \triangleleft \{S \mapsto up_{in}(\mu, S, R_1, \ldots, R_m)\}$ is a TRUB.*

Corollary 17. *When started with a TRUB μ_0, the inner macrostate analysis terminates and returns a TRUB μ^*_{in}.*

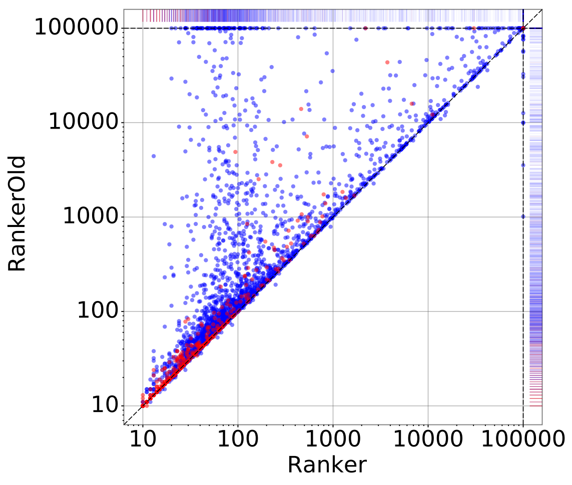
6 Experimental Evaluation

Used tools and evaluation environment. We implemented the techniques described in the previous sections as an extension of the tool Ranker [18] (written in C++). Speaking in the terms of [19], the heuristics were implemented on top of the Ranker$_{\text{MaxR}}$ configuration (we refer to this previous version as Ranker$_{\text{Old}}$). We tested the correctness of our implementation using Spot's `autcross` on all BAs in our benchmark. We compared modified Ranker with other state-of-the-art tools, namely, Goal [41] (implementing Piterman [34], Schewe [37], Safra [36], and Fribourg [1]), Spot 2.9.3 [12] (implementing Redziejowski's algorithm [35]), Seminator 2 [4], LTL2dstar 0.5.4 [23], and Roll [26]. All tools were set to the mode where they output an automaton with the standard state-based Büchi acceptance condition. The experimental evaluation was performed on a 64-bit GNU/Linux Debian workstation with an Intel(R) Xeon(R) CPU E5-2620 running at 2.40 GHz with 32 GiB of RAM and using a timeout of 5 minutes.

Datasets. As the source of our benchmark, we use the two following datasets: (i) `random` containing 11,000 BAs over a two letter alphabet used in [40], which were randomly



(a) Ranker vs Schewe



(b) Ranker vs Ranker$_{\text{Old}}$

Fig. 10: Comparison of the state space generated by our optimizations and other rank-based procedures (horizontal and vertical dashed lines represent timeouts). Blue data points are from `random` and red data points are from `LTL`. Axes are logarithmic.

generated via the Tabakov-Vardi approach [39], starting from 15 states and with various different parameters; (ii) `LTL` with 1,721 BAs over larger alphabets (up to 128 symbols) used in [4], which were obtained from LTL formulae from literature (221) or randomly generated (1,500). We preprocessed the automata using Rabit [30] and Spot's `autfilt` (using the `--high` simplification level), transformed them to state-based acceptance BAs (if they were not already), and converted to the HOA format [2]. From this set, we removed automata that were (i) semi-deterministic, (ii) inherently weak, (iii) unambiguous, or (iv) have an empty language, since for these automata types there exist more efficient complementation procedures than for unrestricted BAs [5,4,6,28]. In the end, we were left with **2,592** (`random`) and **414** (`LTL`) *hard* automata. We use `all` to denote their union (**3,006** BAs). Of these hard automata, 458 were elevator automata.

6.1 Generated State Space

In our first experiment, we evaluated the effectiveness of our heuristics for pruning the generated state space by comparing the sizes of complemented BAs without postprocessing. This use case is directed towards applications where postprocessing is irrelevant, such as inclusion or equivalence checking of BAs.

We focused on a comparison with two less optimized versions of the rank-based complementation procedure: Schewe (the version "Reduced Average Outdegree" from [37] implemented in Goal under `-m rank -tr -ro`) and its optimization Ranker$_{\text{Old}}$. The scatter plots in Fig. 10 compare the numbers of states of automata generated by Ranker and the other algorithms and the upper part of Table 1 gives summary statistics. Observe that our optimizations from this paper drastically reduced the generated search space compared with both Schewe and Ranker$_{\text{Old}}$ (the mean for Schewe is lower than for Ranker$_{\text{Old}}$ due to its much higher number of timeouts); from Fig. 10b we can see that the improvement was in many cases *exponential* even when compared with our previous optimizations in Ranker$_{\text{Old}}$. The median (which is a more meaningful indicator with the presence of timeouts) decreased by 44 % w.r.t. Ranker$_{\text{Old}}$, and we also reduced the

Table 1: Statistics for our experiments. The upper part compares various optimizations of the rank-based procedure (no postprocessing). The lower part compares RANKER to other approaches (with postprocessing). The left-hand side compares sizes of complement BAs and the right-hand side runtimes of the tools. The **wins** and **losses** columns give the number of times when RANKER was strictly better and worse. The values are given for the three datasets as "`all` (`random` : `LTL`)". Approaches in GOAL are labelled with ⚽.

method	mean	median	wins	losses	mean runtime [s]	median runtime [s]	timeouts
RANKER	3812 (4452 : 207)	79 (93 : 26)			7.83 (8.99 : 1.30)	0.51 (0.84 : 0.04)	279 (276 : 3)
RANKER$_{\text{OLD}}$	7398 (8688 : 358)	141 (197 : 29)	2190 (2011 : 179)	111 (107 : 4)	9.37 (10.73 : 1.99)	0.61 (1.04 : 0.04)	365 (360 : 5)
SCHEWE ⚽	4550 (5495 : 665)	439 (774 : 35)	2640 (2315 : 325)	55 (1 : 54)	21.05 (24.28 : 7.80)	6.57 (7.39 : 5.21)	937 (928 : 9)
RANKER	47 (52 : 18)	22 (27 : 10)			7.83 (8.99 : 1.30)	0.51 (0.84 : 0.04)	279 (276 : 3)
PITERMAN ⚽	73 (82 : 22)	28 (34 : 14)	1435 (1124 : 311)	416 (360 : 56)	7.29 (7.39 : 6.65)	5.99 (6.04 : 5.62)	14 (12 : 2)
SAFRA ⚽	83 (91 : 30)	29 (35 : 17)	1562 (1211 : 351)	387 (350 : 37)	14.11 (15.05 : 8.37)	6.71 (6.92 : 5.79)	172 (158 : 14)
SPOT	75 (85 : 15)	24 (32 : 10)	1087 (936 : 151)	683 (501 : 182)	0.86 (0.99 : 0.06)	0.02 (0.02 : 0.02)	13 (13 : 0)
FRIBOURG ⚽	91 (104 : 13)	23 (31 : 9)	1120 (1055 : 65)	601 (376 : 225)	17.79 (19.53 : 7.22)	9.25 (10.15 : 5.48)	81 (80 : 1)
LTL2DSTAR	73 (82 : 21)	28 (34 : 13)	1465 (1195 : 270)	465 (383 : 82)	3.31 (3.84 : 0.11)	0.04 (0.05 : 0.02)	136 (130 : 6)
SEMINATOR 2	79 (91 : 15)	21 (29 : 10)	1266 (1131 : 135)	571 (367 : 204)	9.51 (11.25 : 0.08)	0.22 (0.39 : 0.02)	363 (362 : 1)
ROLL	18 (19 : 14)	10 (9 : 11)	2116 (1858 : 258)	569 (443 : 126)	31.23 (37.85 : 7.28)	8.19 (12.23 : 2.74)	1109 (1106 : 3)

number of timeouts by 23 %. Notice that the numbers for the `LTL` dataset do not differ as much as for `random`, witnessing the easier structure of the BAs in `LTL`.

6.2 Comparison with Other Complementation Techniques

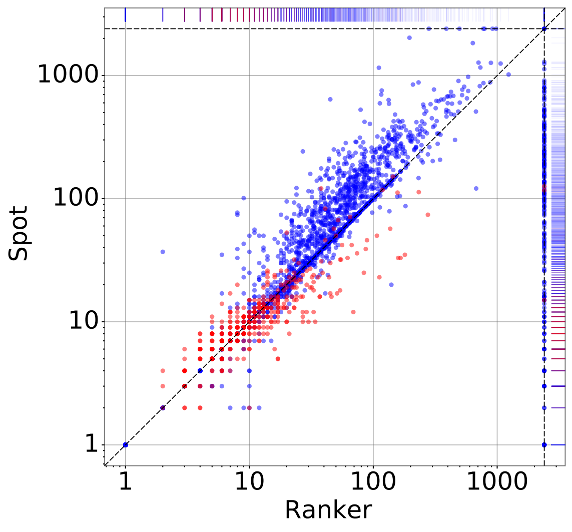
In our second experiment, we compared the improved RANKER with other state-of-the-art tools. We were comparing sizes of output BAs, therefore, we postprocessed each output automaton with `autfilt` (simplification level `--high`). Scatter plots are given in Fig. 11, where we compare RANKER with SPOT (which had the best results on average from the other tools except ROLL) and ROLL, and summary statistics are in the lower part of Table 1. Observe that RANKER has by far the lowest mean (except ROLL) and the third lowest median (after SEMINATOR 2 and ROLL, but with less timeouts). Moreover, comparing the numbers in columns **wins** and **losses** we can see that RANKER gives strictly better results than other tools (**wins**) more often than the other way round (**losses**).

In Fig. 11a see that indeed in the majority of cases RANKER gives a smaller BA than SPOT, especially for harder BAs (SPOT, however, behaves slightly better on the simpler BAs from `LTL`). The results in Fig. 11b do not seem so clear. ROLL uses a learning-based approach—more heavyweight and completely orthogonal to any of the other tools—and can in some cases output a tiny automaton, but does not scale, as observed by the number of timeouts much higher than any other tool. It is, therefore, positively surprising that RANKER could in most of the cases still obtain a much smaller automaton than ROLL.

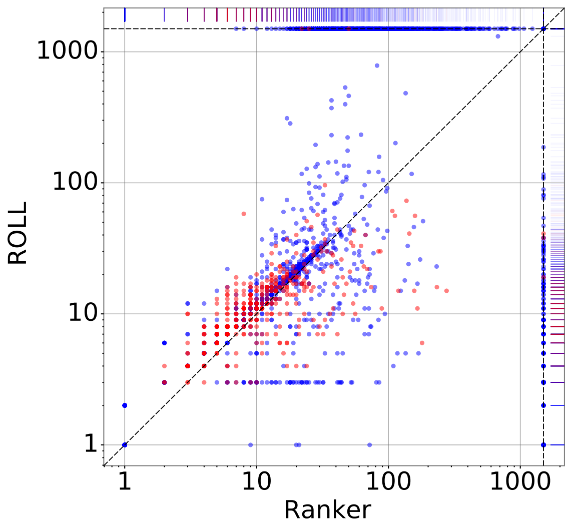
Regarding runtimes, the prototype implementation in RANKER is comparable to SEMINATOR 2, but slower than SPOT and LTL2DSTAR (SPOT is the fastest tool). Implementations of other approaches clearly do not target speed. We note that the number of timeouts of RANKER is still higher than of some other tools (in particular PITERMAN, SPOT, FRIBOURG); further state space reduction targeting this particular issue is our future work.

7 Related Work

BA complementation remains in the interest of researchers since their first introduction by Büchi in [8]. Together with a hunt for efficient complementation techniques, the effort has been put into establishing the lower bound. First, Michel showed that the lower bound is $n!$ (approx. $(0.36n)^n$) [31] and later Yan refined the result to $(0.76n)^n$ [43].



(a) Ranker vs Spot



(b) Ranker vs Roll

Fig. 11: Comparison of the complement size obtained by Ranker and other state-of-the-art tools (horizontal and vertical dashed lines represent timeouts). Axes are logarithmic.

The complementation approaches can be roughly divided into several branches. *Ramsey-based complementation*, the very first complementation construction, where the language of an input automaton is decomposed into a finite number of equivalence classes, was proposed by Büchi and was further enhanced in [7]. *Determinization-based complementation* was presented by Safra in [36] and later improved by Piterman in [34] and Redziejowski in [35]. Various optimizations for determinization of BAs were further proposed in [29]. The main idea of this approach is to convert an input BA into an equivalent deterministic automaton with different acceptance condition that can be easily complemented (e.g. Rabin automaton). The complemented automaton is then converted back into a BA (often for the price of some blow-up). *Slice-based complementation* tracks the acceptance condition using a reduced abstraction on a run tree [42,21]. *A learning-based approach* was introduced in [27,26]. Allred and Ultes-Nitsche then presented a novel optimal complementation algorithm in [1]. For some special types of BAs, e.g., deterministic [25], semi-deterministic [5], or unambiguous [28], there exist specific complementation algorithms. *Semi-determinization based complementation* converts an input BA into a semi-deterministic BA [11], which is then complemented [4].

Rank-based complementation, studied in [24,15,14,37,22], extends the subset construction for determinization of finite automata by storing additional information in each macrostate to track the acceptance condition of all runs of the input automaton. Optimizations of an alternative (sub-optimal) rank-based construction from [24] going through *alternating Büchi automata* were presented in [15]. Furthermore, the work in [22] introduces an optimization of Schewe, in some cases producing smaller automata (this construction is not compatible with our optimizations). As shown in [9], the rank-based construction can be optimized using simulation relations. We identified several heuristics that help reducing the size of the complement in [19], which are compatible with the heuristics in this paper.

Acknowledgements. We thank anonymous reviewers for their useful remarks that helped us improve the quality of the paper. This work was supported by the Czech Science Foundation project 20-07487S and the FIT BUT internal project FIT-S-20-6427.

References

1. Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Proceedings of the Thirty third Annual IEEE Symposium on Logic in Computer Science (LICS 2018). pp. 46–55. IEEE Computer Society Press (July 2018)
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31
3. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.H.: Complementing semi-deterministic büchi automata. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 770–787. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
4. Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV'20). Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (Jul 2020)
5. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.: Complementing semi-deterministic Büchi automata. In: Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 770–787. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_49
6. Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2083, pp. 611–625. Springer (2001). https://doi.org/10.1007/3-540-45744-5_50
7. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Proc. of FOSSACS'12. pp. 150–164. Springer (2012)
8. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. of International Congress on Logic, Method, and Philosophy of Science 1960. Stanford Univ. Press, Stanford (1962)
9. Chen, Y., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 447–467. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_23
10. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). https://doi.org/10.1145/3192366.3192405
11. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 338–345. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21950
12. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Automated Technology for Verification and Analysis. pp. 122–129. Springer International Publishing, Cham (2016)

13. Fogarty, S., Vardi, M.Y.: Büchi complementation and size-change termination. In: Proc. of TACAS'09. pp. 16–30. Springer (2009)
14. Friedgut, E., Kupferman, O., Vardi, M.: Büchi complementation made tighter. International Journal of Foundations of Computer Science **17**, 851–868 (2006)
15. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. LNCS, vol. 2860, pp. 96–110. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_10
16. Havlena, V., Lengál, O., Smahlíková, B.: Sky is not the limit: Tighter rank bounds for elevator automata in Büchi automata complementation (technical report). CoRR **abs/2110.10187** (2021), https://arxiv.org/abs/2110.10187
17. Havlena, V., Lengál, O., Šmahlíková, B.: Deciding S1S: Down the rabbit hole and through the looking glass. In: Proceedings of NETYS'21. pp. 215–222. No. 12754 in LNCS, Springer Verlag (2021). https://doi.org/10.1007/978-3-030-91014-3_15
18. Havlena, V., Lengál, O., Šmahlíková, B.: RANKER (2021), https://github.com/vhavlena/ranker
19. Havlena, V., Lengál, O.: Reducing (To) the Ranks: Efficient Rank-Based Büchi Automata Complementation. In: Proc. of CONCUR'21. LIPIcs, vol. 203, pp. 2:1–2:19. Schloss Dagstuhl, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.2, iSSN: 1868-8969
20. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Proc. of CAV'14. pp. 797–813. Springer (2014)
21. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Proc. of ICALP'08. pp. 724–735. Springer (2008)
22. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Proc. of ATVA'09. LNCS, vol. 5799, pp. 228–243. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_18
23. Klein, J., Baier, C.: On-the-fly stuttering in the construction of deterministic *omega* -automata. In: Proc. of CIAA'07. LNCS, vol. 4783, pp. 51–61. Springer (2007). https://doi.org/10.1007/978-3-540-76336-9_7
24. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993
25. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. **35**(1), 59–71 (1987). https://doi.org/10.1016/0022-0000(87)90036-5
26. Li, Y., Sun, X., Turrini, A., Chen, Y., Xu, J.: ROLL 1.0: ω-regular language learning library. In: Proc. of TACAS'19. LNCS, vol. 11427, pp. 365–371. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_23
27. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Proc. of VMCAI'18. pp. 313–335. Springer (2018)
28. Li, Y., Vardi, M.Y., Zhang, L.: On the power of unambiguity in Büchi complementation. In: Proc. of GandALF'20. EPTCS, vol. 326, pp. 182–198. Open Publishing Association (2020). https://doi.org/10.4204/EPTCS.326.12
29. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of büchi automata. In: Automated Technology for Verification and Analysis. pp. 317–333. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_18
30. Mayr, R., Clemente, L.: Advanced automata minimization. In: Proc. of POPL'13. pp. 63–74 (2013)
31. Michel, M.: Complementation is more difficult with automata on infinite words. CNET, Paris **15** (1988)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6

33. Oei, R., Ma, D., Schulz, C., Hieronymi, P.: Pecan: An automated theorem prover for automatic sequences using büchi automata. CoRR **abs/2102.01727** (2021), https://arxiv.org/abs/2102.01727
34. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proc. of LICS'06. pp. 255–264. IEEE (2006)
35. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744
36. Safra, S.: On the complexity of ω-automata. In: Proc. of FOCS'88. pp. 319–327. IEEE (1988)
37. Schewe, S.: Büchi complementation made tight. In: Proc. of STACS'09. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854
38. Sistla, A.P., Vardi, M.Y., Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. Theoretical Computer Science **49**(2-3), 217–237 (1987)
39. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Proc. of LPAR'05. pp. 396–411. Springer (2005)
40. Tsai, M.H., Fogarty, S., Vardi, M.Y., Tsay, Y.K.: State of Büchi complementation. In: Implementation and Application of Automata. pp. 261–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
41. Tsai, M.H., Tsay, Y.K., Hwang, Y.S.: GOAL for games, omega-automata, and logics. In: Computer Aided Verification. pp. 883–889. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
42. Vardi, M.Y., Wilke, T.: Automata: From logics to algorithms. Logic and Automata **2**, 629–736 (2008)
43. Yan, Q.: Lower bounds for complementation of ω-automata via the full automata technique. In: Automata, Languages and Programming. pp. 589–600. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

Modular Mix-and-Match Complementation of Büchi Automata

Vojtěch Havlena[1](✉), Ondřej Lengál[1](✉), Yong Li[2,3](✉), Barbora Šmahlíková[1](✉), and Andrea Turrini[3,4](✉)

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
ihavlena@fit.vut.cz, lengal@vut.cz, xsmahl00@vut.cz

[2] Department of Computer Science, University of Liverpool, Liverpool, UK
liyong@liverpool.ac.uk

[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, People's Republic of China
turrini@ios.ac.cn

[4] Institute of Intelligent Software, Guangzhou, Guangzhou, People's Republic of China

Abstract. Complementation of nondeterministic Büchi automata (BAs) is an important problem in automata theory with numerous applications in formal verification, such as termination analysis of programs, model checking, or in decision procedures of some logics. We build on ideas from a recent work on BA determinization by Li *et al.* and propose a new modular algorithm for BA complementation. Our algorithm allows to combine several BA complementation procedures together, with one procedure for a subset of the BA's strongly connected components (SCCs). In this way, one can exploit the structure of particular SCCs (such as when they are inherently weak or deterministic) and use more efficient specialized algorithms, regardless of the structure of the whole BA. We give a general framework into which partial complementation procedures can be plugged in, and its instantiation with several algorithms. The framework can, in general, produce a complement with an Emerson-Lei acceptance condition, which can often be more compact. Using the algorithm, we were able to establish an exponentially better new upper bound of $\mathcal{O}(4^n)$ for complementation of the recently introduced class of elevator automata. We implemented the algorithm in a prototype and performed a comprehensive set of experiments on a large set of benchmarks, showing that our framework complements well the state of the art and that it can serve as a basis for future efficient BA complementation and inclusion checking algorithms.

1 Introduction

Nondeterministic Büchi automata (BAs) [8] are an elegant and conceptually simple framework to model infinite behaviors of systems and the properties they are expected to satisfy. BAs are widely used in many important verification tasks, such as termination analysis of programs [30], model checking [54], or as the underlying formal model of decision procedures for some logics (such as S1S [8] or a fragment of the first-order logic over Sturmian words [31]). Many of these applications require to perform *complementation* of BAs: For instance, in termination analysis of programs within Ultimate Automizer [30], complementation is used to keep track of the set of paths whose termination still needs to be proved. On the other hand, in model checking[5] and decision

[5] Here, we consider model checking w.r.t. a specification given in some more expressive logic, such as S1S [8], QPTL [50], or HyperLTL [12], rather than LTL [44], where negation is simple.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 249–270, 2023.
https://doi.org/10.1007/978-3-031-30823-9_13

procedures of logics, complement is usually used to implement negation and quantifier alternation. Complementation is often the most difficult automata operation performed here; its worst-case state complexity is $\mathcal{O}((0.76n)^n)$ [48,2] (which is tight [55]).

In these applications, efficiency of the complementation often determines the overall efficiency (or even feasibility) of the top-level application. For instance, the success of Ultimate Automizer in the Termination category of the International Competition on Software Verification (SV-COMP) [51] is to a large degree due to an efficient BA complementation algorithm [6,11] tailored for BAs with a special structure that it often encounters (as of the time of writing, it has won 6 gold medals in the years 2017–2022 and two silver medals in 2015 and 2016). The special structure in this case are the so-called *semi-deterministic BAs* (SDBAs), BAs consisting of two parts: (i) an initial part without accepting states/transitions and (ii) a deterministic part containing accepting states/transitions that cannot transition into the first part.

Complementation of SDBAs using one from the family of the so-called NCSB algorithms [6,5,11,28] has the worst-case complexity $\mathcal{O}(4^n)$ (and usually also works much better in practice than general BA complementation procedures). Similarly, there are efficient complementation procedures for other subclasses of BAs, e.g., (i) *deterministic BAs* (DBAs) can be complemented into BAs with $2n$ states [35] (or into co-Büchi automata with $n+1$ states) or (ii) *inherently weak BAs* (BAs where in each *strongly connected component* (SCC), either all cycles are accepting or all cycles are rejecting) can be complemented into DBAs with $\mathcal{O}(3^n)$ states using the Miyano-Hayashi algorithm [42].

For a long time, there has been no efficient algorithm for complementation of BAs that are highly structured but do not fall into one of the categories above, e.g., BAs containing inherently weak, deterministic, and some nondeterministic SCCs. For such BAs, one needed to use a general complementation algorithm with the $\mathcal{O}((0.76n)^n)$ (or worse) complexity. To the best of our knowledge, only recently has there appeared works that exploit the structure of BAs to obtain a more efficient complementation algorithm: (i) The work of Havlena *et al.* [29], who introduce the class of *elevator automata* (BAs with an arbitrary mixture of inherently weak and deterministic SCCs) and give a $\mathcal{O}(16^n)$ algorithm for them. (ii) The work of Li *et al.* [37], who propose a BA determinization procedure (into a deterministic Emerson-Lei automaton) that is based on decomposing the input BA into SCCs and using a different determinization procedure for different types of SCCs (inherently weak, deterministic, general) in a synchronous construction.

In this paper, we propose a new BA complementation algorithm inspired by [37], where we exploit the fact that complementation is, in a sense, more relaxed than determinization. In particular, we present a *framework* where one can plug-in different partial complementation procedures fine-tuned for SCCs with a specific structure. The procedures work only with the given SCCs, to some degree *independently* (thus reducing the potential state space explosion) from the rest of the BA. Our top-level algorithm then orchestrates runs of the different procedures in a *synchronous* manner (or completely independently in the so-called *postponed* strategy), obtaining a resulting automaton with potentially a more general acceptance condition (in general an Emerson-Lei condition), which can help keeping the result small. If the procedures satisfy given correctness requirements, our framework guarantees that its instantiation will also be correct. We also propose its optimizations by, e.g., using round-robin to decrease the amount of nondeterminism, using a shared breakpoint to reduce the size and the number of colours for certain class of partial algorithms, and generalize simulation-based pruning of macrostates.

We provide a detailed description of partial complementation procedures for inherently weak, deterministic, and initial deterministic SCCs, which we use to obtain a *new* exponentially better upper bound of $\mathcal{O}(4^n)$ for the class of elevator automata (i.e., the same upper bound as for its strict subclass of SDBAs). Furthermore, we also provide two partial procedures for general SCCs based on determinization (from [37]) and the rank-based construction. Using a prototype implementation, we then show our algorithm complements well existing approaches and significantly improves the state of the art.

2 Preliminaries

We fix a finite non-empty alphabet Σ and the first infinite ordinal ω. An (infinite) word w is a function $w\colon \omega \to \Sigma$ where the i-th symbol is denoted as w_i. Sometimes, we represent w as an infinite sequence $w = w_0 w_1 \dots$ We denote the set of all infinite words over Σ as Σ^ω; an *ω-language* is a subset of Σ^ω.

Emerson-Lei Acceptance Conditions. Given a set $\Gamma = \{0, \dots, k-1\}$ of k *colours* (often depicted as ⓿, ❶, etc.), we define the set of *Emerson-Lei acceptance conditions* $\mathbb{EL}(\Gamma)$ as the set of formulae constructed according to the following grammar:

$$\alpha ::= \mathsf{Inf}(c) \mid \mathsf{Fin}(c) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha) \tag{1}$$

for $c \in \Gamma$. The *satisfaction* relation $\models$ for a set of colours $M \subseteq \Gamma$ and condition α is defined inductively as follows (for $c \in \Gamma$):

$$\begin{aligned} M &\models \mathsf{Fin}(c) \text{ iff } c \notin M, & M &\models \alpha_1 \vee \alpha_2 \text{ iff } M \models \alpha_1 \text{ or } M \models \alpha_2, \\ M &\models \mathsf{Inf}(c) \text{ iff } c \in M, & M &\models \alpha_1 \wedge \alpha_2 \text{ iff } M \models \alpha_1 \text{ and } M \models \alpha_2. \end{aligned}$$

Emerson-Lei Automata. A (nondeterministic transition-based[6]) *Emerson-Lei automaton* (TELA) over Σ is a tuple $\mathcal{A} = (Q, \delta, I, \Gamma, \mathsf{p}, \mathsf{Acc})$, where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*[7], $I \subseteq Q$ is the set of *initial* states, Γ is the set of *colours*, $\mathsf{p}\colon \delta \to 2^\Gamma$ is a *colouring function* of transitions, and $\mathsf{Acc} \in \mathbb{EL}(\Gamma)$. We use $p \xrightarrow{a} q$ to denote that $(p, a, q) \in \delta$ and sometimes also treat δ as a function $\delta\colon Q \times \Sigma \to 2^Q$. Moreover, we extend δ to sets of states $P \subseteq Q$ as $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$. We use $\mathcal{A}[q]$ for $q \in Q$ to denote the automaton $\mathcal{A}[q] = (Q, \delta, \{q\}, \Gamma, \mathsf{p}, \mathsf{Acc})$, i.e., the TELA obtained from $\mathcal{A}$ by setting q as the only initial state. $\mathcal{A}$ is called *deterministic* if $|I| \leq 1$ and $|\delta(q, a)| \leq 1$ for each $q \in Q$ and $a \in \Sigma$. If $\Gamma = \{$⓿$\}$ and $\mathsf{Acc} = \mathsf{Inf}($⓿$)$, we call $\mathcal{A}$ a *Büchi automaton* (BA) and denote it as $\mathcal{A} = (Q, \delta, I, F)$ where F is the set of all transitions coloured by ⓿, i.e., $F = \mathsf{p}^{-1}(\{$⓿$\})$. For a BA, we use $\delta_F(p, a) = \{q \in \delta(p, a) \mid \mathsf{p}(p \xrightarrow{a} q) = \{$⓿$\}\}$ (and extend the notation to sets of states as for δ). A BA $\mathcal{A} = (Q, \delta, I, F)$ is called *semi-deterministic* (SDBA) if for every accepting transition $(p \xrightarrow{a} q) \in F$, the reachable part of $\mathcal{A}[q]$ is deterministic.

A *run* of $\mathcal{A}$ from $q \in Q$ on an input word w is an infinite sequence $\rho\colon \omega \to Q$ that starts in q and respects δ, i.e., $\rho_0 = q$ and $\forall i \geq 0\colon \rho_i \xrightarrow{w_i} \rho_{i+1} \in \delta$. Let $\inf_\delta(\rho) \subseteq \delta$ denote the set of transitions occurring in ρ infinitely often and $\inf_\Gamma(\rho) = \bigcup\{\mathsf{p}(x) \mid x \in$

[6] We only consider transition-based acceptance in order to avoid cluttering the paper by always dealing with accepting states *and* accepting transitions. Extending our approach to state/transition-based (or just state-based) automata is straightforward.

[7] Note that some authors use a more general definition of TELAs with $\delta \subseteq Q \times \Sigma \times 2^\Gamma \times Q$; we only use them as the output of our algorithm, where the simpler definition suffices.

$\inf_{\delta}(\rho)\}$ be the set of infinitely often occurring colours. A run ρ is *accepting* in $\mathcal{A}$ iff $\inf_{\Gamma}(\rho) \models \mathrm{Acc}$ and the *language* of $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$, is defined as the set of words $w \in \Sigma^{\omega}$ for which there exists an accepting run in $\mathcal{A}$ starting with some state in I.

Consider a BA $\mathcal{A} = (Q, \delta, I, F)$. For a set of states $S \subseteq Q$ we use $\mathcal{A}_S$ to denote the copy of $\mathcal{A}$ where accepting transitions only occur between states from S, i.e., the BA $\mathcal{A}_S = (Q, \delta, I, F \cap \delta_{|S})$ where $\delta_{|S} = \{p \xrightarrow{a} q \in \delta \mid p, q \in S\}$. We say that a non-empty set of states $C \subseteq Q$ is a *strongly connected component* (SCC) if every pair of states of C can reach each other and C is a maximal such set. An SCC of $\mathcal{A}$ is *trivial* if it consists of a single state that does not contain a self-loop and *non-trivial* otherwise. An SCC C is *accepting* if it contains at least one accepting transition and *inherently weak* iff either (i) every cycle in C contains a transition from F or (ii) no cycle in C contains any transitions from F. An SCC C is *deterministic* iff the BA $(C, \delta_{|C}, \{q\}, \emptyset)$ for any $q \in C$ is deterministic. We denote inherently weak components as IWCs, accepting deterministic components that are not inherently weak as DACs (deterministic accepting), and the remaining accepting components as NACs (nondeterministic accepting). A BA $\mathcal{A}$ is called an *elevator automaton* if it contains no NAC.

We assume that $\mathcal{A}$ contains no accepting transition outside its SCCs (no run can cycle over such transitions). We use δ_{SCC} to denote the restriction of δ to transitions that do not leave their SCCs, formally, $\delta_{\mathrm{SCC}} = \{p \xrightarrow{a} q \in \delta \mid p \text{ and } q \text{ are in the same SCC}\}$. A *partition block* $P \subseteq Q$ of $\mathcal{A}$ is a nonempty union of its accepting SCCs, and a *partitioning* of $\mathcal{A}$ is a sequence $P_1, \ldots, P_n$ of pairwise disjoint partition blocks of $\mathcal{A}$ that contains all accepting SCCs of $\mathcal{A}$. Given a P_i, let $\mathcal{A}_{P_i}$ be the BA obtained from $\mathcal{A}$ by removing colours from transitions outside P_i. The following fact serves as the basis of our decomposition-based complementation procedure.

Fact 1. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P_1}) \cup \ldots \cup \mathcal{L}(\mathcal{A}_{P_n})$

The complement (automaton) of a BA $\mathcal{A}$ is a TELA that accepts the complement language $\Sigma^{\omega} \setminus \mathcal{L}(\mathcal{A})$ of $\mathcal{L}(\mathcal{A})$. In the paper, we call a state and a run of a complement automaton a *macrostate* and a *macrorun*, respectively.

3 A Modular Complementation Algorithm

In a nutshell, the main idea of our BA complementation algorithm is to first decompose a BA $\mathcal{A}$ into several partition blocks according to their properties, and then perform complementation for each of the partition blocks (potentially using a different algorithm) independently, using either a *synchronous* construction, synchronizing the complementation algorithms for all partition blocks in each step, or a *postponed* construction, which complements the partition blocks independently and combines the partial results using automata product construction. The decomposition of $\mathcal{A}$ into partition blocks can either be trivial—i.e., with one block for each accepting SCC—, or more elaborate, e.g., a partitioning where one partition block contains all accepting IWCs, another contains all DACs, and each NAC is given its own partition block. In this way, one can avoid running a general complementation algorithm for unrestricted BAs with the state complexity upper bound $\mathcal{O}((0.76n)^n)$ and, instead, apply the most suitable complementation procedure for each of the partition blocks. This comes with three main advantages:

1. The complementation algorithm for each partition block can be selected differently in order to exploit the properties of the block. For instance, for partition blocks

with IWCs, one can use complementation based on the breakpoint (the so-called Miyano-Hayashi) construction [42] with $O(3^n)$ macrostates (cf. Sec. 4.1), while for partition blocks with only DACs, one can use an algorithm with the state complexity $O(4^n)$ based on an adaptation of the NCSB construction [6,5,11,28] for SDBAs (cf. Sec. 4.2). For NACs, one can choose between, e.g., rank- [34,21,48,10,24,29] or determinization-based [46,43,45] algorithms, depending on the properties of the NACs (cf. Sec. 6).

2. The different complementation algorithms can focus only on the respective blocks and do not need to consider other parts of the BA. This is advantageous, e.g., for rank-based algorithms, which can use this restriction to obtain tighter bounds on the considered ranks (even tighter than using the refinement in [29]).
3. The obtained automaton can be more compact due to the use of a more general acceptance condition than Büchi [47]—in general, it can be a conjunction of any $\mathbb{EL}$ conditions (one condition for each partition block), depending on the output of the complementation procedures; this can allow a more compact encoding of the produced automaton allowed by using a mixture of conditions. E.g., a deterministic BA can be complemented with constant extra generated states when using a co-Büchi condition rather than a linear number of generated states for a Büchi condition (see Sec. 5.1).

Those partial complementation algorithms then need to be orchestrated by a top-level algorithm to produce the complement of $\mathcal{A}$.

One might regard our algorithm as an optimization of an approach that would for each partition block P obtain a BA $\mathcal{A}_P$, complement $\mathcal{A}_P$ using the selected algorithm, and perform the intersection of all obtained $\mathcal{A}_P$'s (which would, however, not be able to get the upper bound for elevator automata that we give in Sec. 4.3). Indeed, we also implemented the mentioned procedure (called the *postponed* approach, described in Sec. 5.2) and compared it to our main procedure (called the *synchronous* approach).

3.1 Basic Synchronous Algorithm

In this section, we describe the basic *synchronous* top-level algorithm. Then, in Sec. 4, we provide its instantiation for elevator automata and give a new upper bound for their complementation; in Sec. 5, we discuss several optimizations of the algorithm; and in Sec. 6, we give a generalization for unrestricted BAs. Let us fix a BA $\mathcal{A} = (Q, \delta, I, F)$ and, w.l.o.g., assume that $\mathcal{A}$ is *complete*, i.e., $|I| > 0$ and all states $q \in Q$ have an outgoing transition over all symbols $a \in \Sigma$.

The synchronous algorithm works with partial complementation algorithms for BA's partition blocks. Each such algorithm `Alg` is provided with a structural condition $\varphi_{\mathtt{Alg}}$ characterizing partition blocks it can complement. For a BA $\mathcal{B}$, we use the notation $\mathcal{B} \models \varphi$ to denote that $\mathcal{B}$ satisfies the condition φ. We say that `Alg` is a *partial complementation algorithm for a partition block* P if $\mathcal{A}_P \models \varphi_{\mathtt{Alg}}$. We distinguish between `Alg`, a general algorithm able to complement a partition block of a given type, and $\mathtt{Alg}_P$, its instantiation for the partition block P. Each instance $\mathtt{Alg}_P$ is required to provide the following:

- $\mathtt{T}^{\mathtt{Alg}_P}$ — the type of the macrostates produced by the algorithm;
- $\mathtt{Colours}^{\mathtt{Alg}_P} = \{0, \ldots, k^{\mathtt{Alg}_P} - 1\}$ — the set of used colours;
- $\mathtt{Init}^{\mathtt{Alg}_P} \in 2^{\mathtt{T}^{\mathtt{Alg}_P}}$ — the set of initial macrostates;
- $\mathtt{Succ}^{\mathtt{Alg}_P} \colon (2^Q \times \mathtt{T}^{\mathtt{Alg}_P} \times \Sigma) \to 2^{\mathtt{T}^{\mathtt{Alg}_P} \times \mathtt{Colours}^{\mathtt{Alg}_P}}$ — a function returning the successors of a macrostate such that $\mathtt{Succ}^{\mathtt{Alg}_P}(H, M, a) = \{(M_1, \alpha_1), \ldots, (M_k, \alpha_k)\}$, where H is the set of all states of $\mathcal{A}$ reached over the same word, M is the $\mathtt{Alg}_P$'s

macrostate for the given partition block, a is the input symbol, and each (M_i, α_i) is a pair (*macrostate*, *set of colours*) such that M_i is a successor of M over a w.r.t. H and α_i is a set of colours on the edge from M to M_i (H helps to keep track of *new* runs coming into the partition block); and
- $\mathtt{Acc}^{\mathtt{Alg}_P} \in \mathbb{EL}(\mathtt{Colours}^{\mathtt{Alg}_P})$ — the acceptance condition.

Let $P_1, \ldots, P_n$ be a partitioning of $\mathcal{A}$ (w.l.o.g., we assume that $n > 0$), and $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ be a sequence of algorithms such that $\mathtt{Alg}^i$ is a partial complementation algorithm for P_i. Furthermore, let us define the following auxiliary *renumbering* function λ as $\lambda(c, j) = c + \sum_{i=1}^{j-1} |\mathtt{Colours}^{\mathtt{Alg}^i_{P_i}}|$, which is used to make the colours and acceptance conditions from the partial complementation algorithms disjoint. We also lift λ to sets of colours in the natural way, and also to $\mathbb{EL}$ conditions such that $\lambda(\varphi, j)$ has the same structure as φ but each atom $\mathsf{Inf}(c)$ is substituted with the atom $\mathsf{Inf}(\lambda(c, j))$ (and likewise for Fin atoms). The synchronous complementation algorithm then produces the TELA $\textsc{ModCompl}(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A}) = (Q^C, \delta^C, I^C, \Gamma^C, \mathsf{p}^C, \mathsf{Acc}^C)$ with components defined as follows (we use $[S_i]_{i=1}^n$ to abbreviate $S_1 \times \cdots \times S_n$):
- $Q^C = 2^Q \times [\mathtt{T}^{\mathtt{Alg}^i_{P_i}}]_{i=1}^n$,
- $I^C = \{I\} \times [\mathtt{Init}^{\mathtt{Alg}^i_{P_i}}]_{i=1}^n$,
- $\Gamma^C = \{0, \ldots, \lambda(k^{\mathtt{Alg}^n_{P_n}} - 1, n)\}$,
- $\mathsf{Acc}^C = \bigwedge_{i=1}^n \lambda(\mathtt{Acc}^{\mathtt{Alg}^i_{P_i}}, i)$,[8] and
- δ^C and p^C are defined such that if
$$((M'_1, \alpha_1), \ldots, (M'_n, \alpha_n)) \in [\mathtt{Succ}^{\mathtt{Alg}^i_{P_i}}(H, M_i, a)]_{i=1}^n,$$
then δ^C contains the transition $t\colon (H, M_1, \ldots, M_n) \xrightarrow{a} (\delta(H, a), M'_1, \ldots, M'_n)$, coloured by $\mathsf{p}^C(t) = \bigcup\{\lambda(\alpha_i, i) \mid 1 \leq i \leq n\}$, and δ^C is the smallest such a set.

In order for ModCompl to be correct, the partial complementation algorithms need to satisfy certain properties, which we discuss below.

For a structural condition φ and a BA $\mathcal{B} = (Q, \delta, I, F)$, we define $\mathcal{B} \models_P \varphi$ iff $\mathcal{B} \models \varphi$, P is a partition block of $\mathcal{B}$, and $\mathcal{B}$ contains no accepting transitions outside P. We can now provide the correctness condition on $\mathtt{Alg}$.

Definition 1. *We say that* $\mathtt{Alg}$ *is* correct *if for each BA* $\mathcal{B}$ *and partition block* P *such that* $\mathcal{B} \models_P \varphi_{\mathtt{Alg}}$ *it holds that* $\mathcal{L}(\textsc{ModCompl}(\mathtt{Alg}_P, \mathcal{B})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.

The correctness of the synchronous algorithm (provided that each partial complementation algorithm is correct) is then established by Theorem 1.

Theorem 1. *Let* $\mathcal{A}$ *be a BA,* $P_1, \ldots, P_n$ *be a partitioning of* $\mathcal{A}$*, and* $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ *be a sequence of partial complementation algorithms such that* $\mathtt{Alg}^i$ *is* correct *for* P_i*. Then, we have* $\mathcal{L}(\textsc{ModCompl}(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.

4 Modular Complementation of Elevator Automata

In this section, we first give partial algorithms to complement partition blocks with only accepting IWCs (Sec. 4.1) and partition blocks with only DACs (Sec. 4.2). Then, in Sec. 4.3, we show that using our algorithm, the upper bound on the size of the complement of elevator BAs is in $\mathcal{O}(4^n)$, which is *exponentially better* than the known upper bound $\mathcal{O}(16^n)$ established in [29].

[8] If we drop the condition that $\mathcal{A}$ is complete, we also need to add an *accepting sink state* (representing the case for $H = \emptyset$) with self-loops over all symbols marked by a new colour Ⓢ, and enrich Acc^C with $\ldots \vee \mathsf{Inf}(Ⓢ)$.

4.1 Complementation of Inherently Weak Accepting Components

First, we introduce a partial algorithm MH with the condition $\varphi_{\mathtt{MH}}$ specifying that all SCCs in the partition block P are *accepting* IWCs. Let P be a partition block of $\mathcal{A}$ such that $\mathcal{A}_P \models \varphi_{\mathtt{MH}}$. Our proposed approach makes use of the Miyano-Hayashi construction [42]. Since in accepting IWCs, all runs are accepting, the idea of the construction is to accept words such that all runs over the words eventually leave P.

Therefore, we use a pair (C, B) of sets of states as a macrostate for complementing P. Intuitively, we use C to denote the set of all runs of $\mathcal{A}$ that are in P (C for "*check*"). The set $B \subseteq C$ represents the runs being inspected whether they leave P at some point (B for "*breakpoint*"). Initially, we let $C = I \cap P$ and also sample into breakpoint all runs in P, i.e., set $B = C$. Along reading an ω-word w, if all runs that have entered P eventually leave P, i.e., B becomes empty infinitely often, the complement language of P should contain w (when B becomes empty, we sample B with all runs from the current C). We formalize $\mathtt{MH}_P$ as a partial procedure in the framework from Sec. 3.1 as follows:

- $\mathtt{T}^{\mathtt{MH}_P} = 2^P \times 2^P$, $\mathtt{Colours}^{\mathtt{MH}_P} = \{\textcircled{0}\}$, $\mathtt{Init}^{\mathtt{MH}_P} = \{(I \cap P, I \cap P)\}$,
- $\mathtt{Acc}^{\mathtt{MH}_P} = \mathsf{Inf}(\textcircled{0})$, and $\mathtt{Succ}^{\mathtt{MH}_P}(H, (C, B), a) = \{((C', B'), \alpha)\}$ where
  - $C' = \delta(H, a) \cap P$,
  - $B' = \begin{cases} C' & \text{if } B^\star = \emptyset \text{ for } B^\star = \delta(B, a) \cap C', \\ B^\star & \text{otherwise, and} \end{cases}$
  - $\alpha = \begin{cases} \{\textcircled{0}\} & \text{if } B^\star = \emptyset \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

We can see that checking whether w is accepted by the complement of P reduces to check whether B has been cleared infinitely often. Since every time when B becomes empty, we emit the colour $\textcircled{0}$, we have that w is not accepted by $\mathcal{A}$ within P if and only if $\textcircled{0}$ occurs infinitely often. Note that the transition function $\mathtt{Succ}^{\mathtt{MH}_P}$ is deterministic, i.e., there is exactly one successor.

Lemma 1. *The partial algorithm* MH *is correct.*

4.2 Complementation of Deterministic Accepting Components

In this section, we give a partial algorithm CSB with the condition $\varphi_{\mathtt{CSB}}$ specifying that a partition block P consists of *DACs*. Let P be a partition block of $\mathcal{A}$ such that $\mathcal{A}_P \models \varphi_{\mathtt{CSB}}$. Our approach is based on the NCSB family of algorithms [6,11,5,28] for complementing SDBAs, in particular the NCSB-MaxRank construction [28]. The algorithm utilizes the fact that runs in DACs are deterministic, i.e., they do not branch into new runs. Therefore, one can check that a run is non-accepting if there is a time point from which the run does not see accepting transitions any more. We call such a run that does not see accepting transitions any more *safe*. Then, an ω-word w is not accepted in P iff all runs over w in P either (i) leave P or (ii) eventually become safe.

For checking point (i), we can use a similar technique as in algorithm MH, i.e., use a pair (C, B). Moreover, to be able to check point (ii), we also use the set S that contains runs that are supposed to be *safe*, resulting in macrostates of the form (C, S, B)[9]. To make sure that all runs are deterministic, we will use δ_{SCC} instead of δ when computing the successors of S and B since there may be nondeterministic jumps between different DACs in P; we will not miss any run in P since if a run moves between DACs of P, it

[9] In contrast to MH, here we use $C \cup S$ rather than C to keep track of all runs in P.

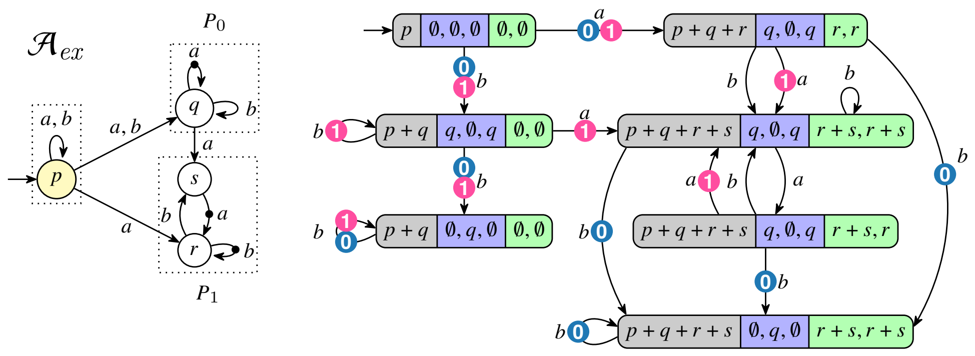

$\mathcal{A}_{ex}$
P_0
P_1
p
q
s
r
a, b
p | $\emptyset, \emptyset, \emptyset$ | $\emptyset, \emptyset$
$p+q+r$ | $q, \emptyset, q$ | r, r
$p+q$ | $q, \emptyset, q$ | $\emptyset, \emptyset$
$p+q+r+s$ | $q, \emptyset, q$ | $r+s, r+s$
$p+q$ | $\emptyset, q, \emptyset$ | $\emptyset, \emptyset$
$p+q+r+s$ | $q, \emptyset, q$ | $r+s, r$
$p+q+r+s$ | $\emptyset, q, \emptyset$ | $r+s, r+s$

Fig. 1: Left: BA $\mathcal{A}_{ex}$ (dots represent accepting transitions). Right: the outcome of ModCompl($\mathtt{CSB}_{P_0}, \mathtt{MH}_{P_1}, \mathcal{A}_{ex}$) with Acc: Inf(⓿) ∧ Inf(❶). States are given as $(H, (C_0, S_0, B_0), (C_1, B_1))$; to avoid too many braces, sets are given as sums.

can be seen as the run leaving P and a new run entering P. Since a run eventually stays in one SCC, this guarantees that the run will not be missed.

We formalize $\mathtt{CSB}_P$ in the top-level framework as follows:

- $\mathtt{T}^{\mathtt{CSB}_P} = 2^P \times 2^P \times 2^P$, $\mathtt{Init}^{\mathtt{CSB}_P} = \{(I \cap P, \emptyset, I \cap P)\}$,
- $\mathtt{Colours}^{\mathtt{CSB}_P} = \{$⓿$\}$, $\mathtt{Acc}^{\mathtt{CSB}_P} = \mathsf{Inf}($⓿$)$, and
- $\mathtt{Succ}^{\mathtt{CSB}_P}(H, (C, S, B), a) = U$ such that
  - if $\delta_F(S, a) \neq \emptyset$, then $U = \emptyset$ (Runs in S must be *safe*),
  - otherwise U contains $((C', S', B'), c)$ where
    - ∗ $S' = \delta_{\mathrm{SCC}}(S, a) \cap P$, $C' = (\delta(H, a) \cap P) \setminus S'$,
    - ∗ $B' = \begin{cases} C' & \text{if } B^\star = \emptyset \text{ for } B^\star = \delta_{\mathrm{SCC}}(B, a), \\ B^\star & \text{otherwise, and} \end{cases}$ ∗ $c = \begin{cases} \{⓿\} & \text{if } B^\star = \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$

    Moreover, in the case $\delta_F(B, a) = \emptyset$, then U also contains $((C'', S'', C''), \{$⓿$\})$ where $S'' = S' \cup B'$ and $C'' = C' \setminus S''$.

Intuitively, when $\delta_F(B, a) \cap \delta_{\mathrm{SCC}}(B, a) = \emptyset$, we make the following guess: (i) either the runs in B all become safe (we move them to S) or (ii) there might be some unsafe runs (we keep them in B). Since the runs in B are deterministic, the number of tracked runs in B will not increase. Moreover, if all runs in B are eventually safe, we are guaranteed to move all of them to S at the right time point, e.g., the maximal time point where all runs are safe since the number of runs is finite.

As mentioned above, w is not accepted within P iff all runs over w either (i) leave P or (ii) become safe. In the context of the presented algorithm, this corresponds to (i) B becoming empty infinitely often and (ii) $\delta_F(S, a)$ never seeing an accepting transition. Then we only need to check if there exists an infinite sequence of macrostates $\hat{\rho} = (C_0, S_0, B_0) \ldots$ that emits ⓿ infinitely often.

Lemma 2. *The partial algorithm* CSB *is correct.*

It is worth noting that when the given partition block P contains all DACs of $\mathcal{A}$, we can still use the construction above, while the construction in [28] only works on SDBAs.

Example 1. In Fig. 1, we give an example of the run of our algorithm on the BA $\mathcal{A}_{ex}$. The BA contains three SCCs, one of them (the one containing p) non-accepting (therefore, it does not need to occur in any partition block). The partition block P_0 contains a single DAC, so we can use algorithm CSB, and the partition block P_1 contains a single accepting IWC, so we can use MH. The resulting ModCompl$(\mathtt{CSB}_{P_0}, \mathtt{MH}_{P_1}, \mathcal{A}_{ex})$ uses two colours, ⓿ from CSB and ❶ from MH. The acceptance condition is $\mathsf{Inf}($⓿$) \wedge \mathsf{Inf}($❶$)$. $\square$

4.3 Upper-bound for Elevator Automata Complementation

We now give an upper bound on the size of the complement generated by our algorithm for elevator automata, which significantly improves the best previously known upper bound of $O(16^n)$ [29] to $O(4^n)$, the same as for SDBAs, which are a strict subclass of elevator automata [6] (we note that this upper bound cannot be obtained by a determinization-based algorithm, since determinization of SDBAs is in $\Omega(n!)$ [17,40]).

Theorem 2. *Let $\mathcal{A}$ be an elevator automaton with n states. Then there exists a BA with $O(4^n)$ states accepting the complement of $\mathcal{L}(\mathcal{A})$.*

Proof (Sketch). Let Q_W be all states in accepting IWCs, Q_D be all states in DACs, and Q_N be the remaining states, i.e., $Q = Q_W \uplus Q_D \uplus Q_N$. We make two partition blocks: $P_0 = Q_W$ and $P_1 = Q_D$ and use MH and CSB respectively as the partial algorithms, with macrostates of the form $(H, (C_0, B_0), (C_1, S_1, B_1))$. For each state $q_N \in Q_N$, there are two options: either $q_N \notin H$ or $q_N \in H$. For each state $q_W \in Q_W$, there are three options: (i) $q_W \notin C_0$, (ii) $q_W \in C_0 \setminus B_0$, or (iii) $q_W \in C_0 \cap B_0$. Finally, for each $q_D \in Q_D$, there are four options: (i) $q_D \notin C_1 \cup S_1$, (ii) $q_D \in S_1$, (iii) $q_D \in C_1 \setminus B_1$, or (iv) $q_D \in C_1 \cap B_1$. Therefore, the total number of macrostates is $2 \cdot 2^{|Q_N|} \cdot 3^{|Q_W|} \cdot 4^{|Q_D|} \in O(4^n)$ where the initial factor 2 is due to degeneralization from two to one colour (the two colours can actually be avoided by using our shared breakpoint optimization from Sec. 5.4). $\square$

5 Optimizations of the Modular Construction

In this section, we propose optimizations of the basic modular algorithm. In Sec. 5.1, we give a partial algorithm to complement initial partition blocks with DACs. Further, in Sec. 5.2, we propose the postponed construction allowing to use automata reduction on intermediate results. In Sec. 5.3, we propose the round-robin algorithm alleviating the problem with the explosion of the size of the Cartesian product of partial successors. In Sec. 5.4, we provide an optimization for partial algorithms that are based on the breakpoint construction, and, finally, in Sec. 5.5, we show how to employ simulation to decrease the size of macrostates in the synchronous construction.

5.1 Complementation of Initial Deterministic Partition Blocks

Our first optimization is an algorithm CoB for a subclass of partition blocks containing DACs. In particular, the condition $\varphi_{\mathtt{CoB}}$ specifies that the partition block P is deterministic and can be reached only deterministically in $\mathcal{A}$ (i.e., $\mathcal{A}_P$ after removing redundant states is deterministic). Then, we say that P is an *initial deterministic* partition block. The algorithm is based on complementation of deterministic BAs into co-Büchi automata.

The algorithm $\mathtt{CoB}_P$ is formalized below:

– $\mathtt{T}^{\mathtt{CoB}_P} = P \cup \{\emptyset\}$, $\mathtt{Init}^{\mathtt{CoB}_P} = I \cap P$, $\mathtt{Colours}^{\mathtt{CoB}_P} = \{$⓿$\}$, $\mathtt{Acc}^{\mathtt{CoB}_P} = \mathsf{Fin}($⓿$)$,

– $\mathtt{Succ}^{\mathtt{CoB}_P}(H, q, a) = \{(q', \alpha)\}$ where
  - $q' = \begin{cases} r & \text{if } \delta(H,a) \cap P = \{r\} \text{ and} \\ \emptyset & \text{otherwise,} \end{cases}$
  - $\alpha = \begin{cases} \{\textcircled{0}\} & \text{if } q \xrightarrow{a} q' \in F \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

Intuitively, all runs reach P deterministically, which means that over a word w, at most one run can reach P (so $|\mathtt{Init}^{\mathtt{CoB}_P}| = 1$). Thus, we have $|\delta(H, w_j) \cap P| = 1$ for some $j \geq 0$ if there is a run over w to P, corresponding to $\delta(H, a) \cap P = \{r\}$ in the construction. To check whether w is not accepted in P, we only need to check whether the run from $r \in P$ over w visits accepting transitions only finitely often. We give an example of complementation of a BA containing an initial deterministic partition block in [27].

Lemma 3. *The partial algorithm* CoB *is correct.*

5.2 Postponed Construction

The modular synchronous construction from Sec. 3.1 utilizes the assumption that in the simultaneous construction of successors for each partition block over a, if one partial macrostate M_i does not have a successor over a, then there will be no successor of the $(H, M_1, \ldots, M_n)$ macrostate in δ^C as well. This is useful, e.g., for inclusion testing, where it is not necessary to generate the whole complement. On the other hand, if we need to generate the whole automaton, a drawback of the proposed modular construction is that each partial complementation algorithm itself may generate a lot of useless states. In this section, we propose the *postponed construction*, which complements the partition blocks (with their surrounding) independently and later combines the intermediate results to obtain the complement automaton for $\mathcal{A}$. The main advantage of the postponed construction is that one can apply automata reduction (e.g., based on removing useless states or using simulation [13,18,1,9]) to decrease the size of the intermediate automata.

In the postponed construction, we use product-based BA intersection operation (i.e., for two TELAs $\mathcal{B}_1$ and $\mathcal{B}_2$, a product automaton $\mathcal{B}_1 \cap \mathcal{B}_2$ satisfying $\mathcal{L}(\mathcal{B}_1 \cap \mathcal{B}_2) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$[10]). Further, we employ a function Red performing some language-preserving reduction of an input TELA. Then, the postponed construction for an elevator automaton $\mathcal{A}$ with a partitioning $P_1, \ldots, P_n$ and a sequence $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ where $\mathtt{Alg}^i$ is a partial complementation algorithm for P_i, is defined as follows:

$$\textsc{PostpCompl}(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A}) = \bigcap_{i=1}^{n} \mathtt{Red}\left(\textsc{ModCompl}(\mathtt{Alg}^i_{P_i}, \mathcal{A}_{P_i})\right). \qquad (2)$$

The correctness of the construction is then summarized by the following theorem.

Theorem 3. *Let $\mathcal{A}$ be a BA, $P_1, \ldots, P_n$ be a partitioning of $\mathcal{A}$, and $\mathtt{Alg}^1, \ldots, \mathtt{Alg}^n$ be a sequence of partial complementation algorithms such that $\mathtt{Alg}^i$ is* correct *for P_i. Then, $\mathcal{L}(\textsc{PostpCompl}(\mathtt{Alg}^1_{P_1}, \ldots, \mathtt{Alg}^n_{P_n}, \mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

5.3 Round-Robin Algorithm

The proposed basic synchronous approach from Sec. 3.1 may suffer from the combinatorial explosion because the successors of a macrostate are given by the Cartesian product of all successors of the partial macrostates. To alleviate this explosion, we propose

[10] Alternatively, one might also avoid the product and generate linear-sized *alternating* TELA, but working with those is usually much harder and not used in practice.

a *round-robin* top-level algorithm. Intuitively, the round-robin algorithm actively tracks runs in only one partial complementation algorithm at a time (while other algorithms stay passive). The algorithm periodically changes the active algorithm to avoid starvation (the decision to leave the active state is, however, fully directed by the partial complementation algorithm). This can alleviate an explosion in the number of successors for algorithms that generate more than one successor (e.g., for rank-based algorithms where one needs to make a nondeterministic choice of decreasing ranks of states in order to be able to accept [34,21,48,10,24,29]; such a choice needs to be made only in the active phase while in the passive phase, the construction just needs to make sure that the run is consistent with the given ranking, which can be done deterministically).

The round-robin algorithm works on the level of *partial complementation round-robin algorithms*. Each instance of the partial algorithm provides *passive types* to represent partial macrostates that are passive and *active types* to represent currently active partial macrostates. In contrast to the basic partial complementation algorithms from Sec. 3.1, which provide only a single successor function, the round-robin partial algorithms provide several variants of them. In particular, `SuccPass` returns (passive) successors of a passive partial macrostate, `Lift` gives all possible active counterparts of a passive macrostate, and `SuccAct` returns successors of an active partial macrostate. If `SuccAct` returns a partial macrostate of the passive type, the round-robin algorithm promotes the next partial algorithm to be the active one. For instance, in the round-robin version of `CSB`, the passive type does not contain the breakpoint and only checks that safe runs stay safe, so it is deterministic. Due to space limitations, we give a formal definition and more details about the round-robin algorithm in [27].

5.4 Shared Breakpoint

The partial complementation algorithms `CSB` and `MH` (and later `RNK` defined in Sec. 6) use a breakpoint to check whether the runs under inspection are accepting or not. As an optimization, we consider merging of breakpoints of several algorithms and keeping only a single breakpoint for all supported algorithms. The top-level algorithm then needs to manage only one breakpoint and emit a colour only if this sole breakpoint becomes empty. This may lead to a smaller number of generated macrostates since we synchronize the breakpoint sampling among several algorithms. The second benefit is that this allows us to generate fewer colours (in the case of elevator automata complemented using algorithms `CSB` and `MH`, we get only one colour).

5.5 Simulation Pruning

Our construction can be further optimized by a simulation (or other compatible) relation for pruning macrostates.[11] A simulation is, broadly speaking, a relation $\preccurlyeq\ \subseteq Q \times Q$ implying language inclusion of states, i.e., $\forall p, q \in Q\colon p \preccurlyeq q \implies \mathcal{L}(\mathcal{A}[p]) \subseteq \mathcal{L}(\mathcal{A}[q])$. Intuitively, our optimization allows to remove a state p from a macrostate M if there is also a state q in M such that (i) $p \preccurlyeq q$, (ii) p is not reachable from q, and (iii) p is smaller than q in an arbitrary total order over Q (this serves as a tie-breaker for

[11] This optimization can be seen as a generalization of the simulation-based pruning techniques that appeared, e.g., in [41,28] in the context of concrete determinization/complementation procedures. Here, we generalize the technique to all procedures that are based on run tracking.

simulation-equivalent mutually unreachable states). The reason why p can be removed is that its behaviour can be completely mimicked by q. In our construction, we can then, roughly speaking, replace each call to the functions $\delta(U, a)$ and $\delta_F(U, a)$, for a set of states U, by $pr(\delta(U, a))$ and $pr(\delta_F(U, a))$ respectively in each partial complementation algorithm, as well as in the top-level algorithm, where $pr(S)$ is obtained from S by pruning all eligible states. The details are provided in [27].

6 Modular Complementation of Non-Elevator Automata

A non-elevator automaton $\mathcal{A}$ contains at least one NAC, besides possibly other IWCs or DACs. To complement $\mathcal{A}$ in a modular way, we apply the techniques seen in Sec. 4 to its DACs and IWCs, while for its NACs we resort to a general complementation algorithm `Alg`. In theory, rank- [34], slice- [32], Ramsey- [50], subset-tuple- [2], and determinization- [46] based complementation algorithms adapted to work on a single partition block instead of the whole automaton are all valid instantiations of `Alg`. Below, we give a high-level description of two such algorithms: rank- and determinization-based.

Rank-based partial complementation algorithm. Working on each NAC independently benefits the complementation algorithm even if the input BA contains only NACs. For instance, in rank-based algorithms [34,21,48,33,10,24,29], the fact whether all runs of $\mathcal{A}$ over a given ω-word w are non-accepting is determined by *ranks* of states, given by the so-called *ranking functions*. A ranking function is a (partial) function from Q to ω. The main idea of rank-based algorithms is the following: (i) every run is initially nondeterministically assigned a rank, (ii) ranks can only decrease along a run, (iii) ranks need to be even every time a run visits an accepting transition, and (iv) the complement automaton accepts iff all runs eventually get trapped in odd ranks[12]. In the standard rank-based procedure, the initial assignment of ranks to states in (i) is a function $Q \rightharpoonup \{0, \dots, 2n-1\}$ for $n = |Q|$. Using our framework, we can, however, significantly restrict the considered ranks in a partition block P to only $P \rightharpoonup \{0, \dots, 2m-1\}$ for $m = |P|$ (here, it makes sense to use partition blocks consisting of single SCCs). One can further reduce the considered ranks using the techniques introduced in, e.g., [24,29].

In order to adapt the rank-based construction as a partial complementation algorithm `RNK` in our framework, we need to extend the ranking functions by a fresh "box state" $\blacksquare$ representing states outside the partition block. The ranking function then uses $\blacksquare$ to represent ranks of runs newly coming into the partition block. The box-extension also requires to change the transition in a way that $\blacksquare$ always represents reachable states from the outside. We provide the details of the construction, which includes the MaxRank optimization from [24], in [27].

Determinization-based partial complementation algorithm. In [52,29] we can see that determinization-based complementation is also a good instantiation of `Alg` in practice, so, we also consider the standard Safra-Piterman determinization [46,43,45] as a choice of `Alg` for complementing NACs. Determinization-based algorithms use a layered subset construction to organize all runs over an ω-word w. The idea is to identify a subset $S \subseteq H$ of reachable states that occur infinitely often along reading w such that between every two occurrences of S, we have that (i) every state in the second occurrence of S can be reached

[12] Since we focus on intuition here, we use runs rather than the directed acyclic graphs of runs.

Table 1: Statistics for our experiments. The column **unsolved** classifies unsolved instances by the form *timeouts : out of memory : other failures*. For the cases of VBS we provide just the number of unsolved cases. The columns **states** and **runtime** provide *mean : median* of the number of states and runtime, respectively.

tool	**solved**	**unsolved**	**states**	**runtime**
KOFOLA$_S$	39,738	89 : 10 : 0	76 : 3	0.32 : 0.03
KOFOLA$_P$	39,750	76 : 11 : 0	86 : 3	0.41 : 0.03
VBS$_+$	39,834	3	78 : 3	0.05 : 0.01
VBS$_-$	39,834	3	96 : 3	0.05 : 0.01
COLA	39,814	21 : 0 : 2	80 : 3	0.17 : 0.02
RANKER	38,837	61 : 939 : 0	45 : 4	3.31 : 0.01
SEMINATOR	39,026	238 : 573 : 0	247 : 3	1.98 : 0.03
SPOT	39,827	8 : 0 : 2	160 : 4	0.08 : 0.02

by a state in the first occurrence of S and (ii) every state in the second occurrence is reached by a state in the first occurrence while seeing an accepting transition. According to König's lemma, there must then be an accepting run of $\mathcal{A}$ over w.

The construction initially maintains only one set H: the set of reachable states. Since S as defined does not necessarily need to be H, every time there are runs visiting accepting transitions, we create a new subset C for those runs and remember which subset C is coming from. This way, we actually organize the current states of all runs into a tree structure and do subset construction in parallel for the sets in each tree node. If we find a tree node whose labelled subset, say S', is equal to the union of states in its children, we know the set S' satisfies the condition above and we remove all its child nodes and emit a good event. If such good event happens infinitely often, it means that S' also occurs infinitely often. So in complementation, we only need to make sure those good events only happen for finitely many times. Working on each NAC separately also benefits the determinization-based approach since the number of possible trees will be less with smaller number of reachable states. Following the idea of [37], to adapt for the construction as the partial complementation algorithm, we put all the newly coming runs from other partition blocks in a newly created node without a parent node. In this way, we actually maintain a forest of trees for the partial complementation construction. We denote the determinization-based construction as DET; cf. [37] for details.

7 Experimental Evaluation

To evaluate the proposed approach, we implemented it in a prototype tool KOFOLA [25] (written in C++) built on top of SPOT [16] and compared it against COLA [37], RANKER [28] (v. 2), SEMINATOR [5] (v. 2.0), and SPOT [15,16] (v. 2.10.6), which are the state of the art in BA complementation [29,28,37]. Due to space restrictions, we give results for only two instantiations of our framework: KOFOLA$_S$ and KOFOLA$_P$. Both instantiations use MH for IWCs, CSB for DACs, and DET for NACs. The partitioning selection algorithm merges all IWCs into one partition block, all DACs into one partition block, and keeps all NACs separate. Simulation-based pruning from Sec. 5.5 is turned on, and round-robin from Sec. 5.3 is turned off (since the selected algorithms are quite deterministic). KOFOLA$_S$ employs the *synchronous* and KOFOLA$_P$ employs the *postponed* strategy. We also consider the Virtual Best Solver (VBS), i.e., a virtual tool that would choose the best solver for each single benchmark among all tools (VBS$_+$) and among all tools except both versions of KOFOLA (VBS$_-$). We ran our experiments on an Ubuntu 20.04.4 LTS system running on a desktop machine with 16 GiB RAM and an

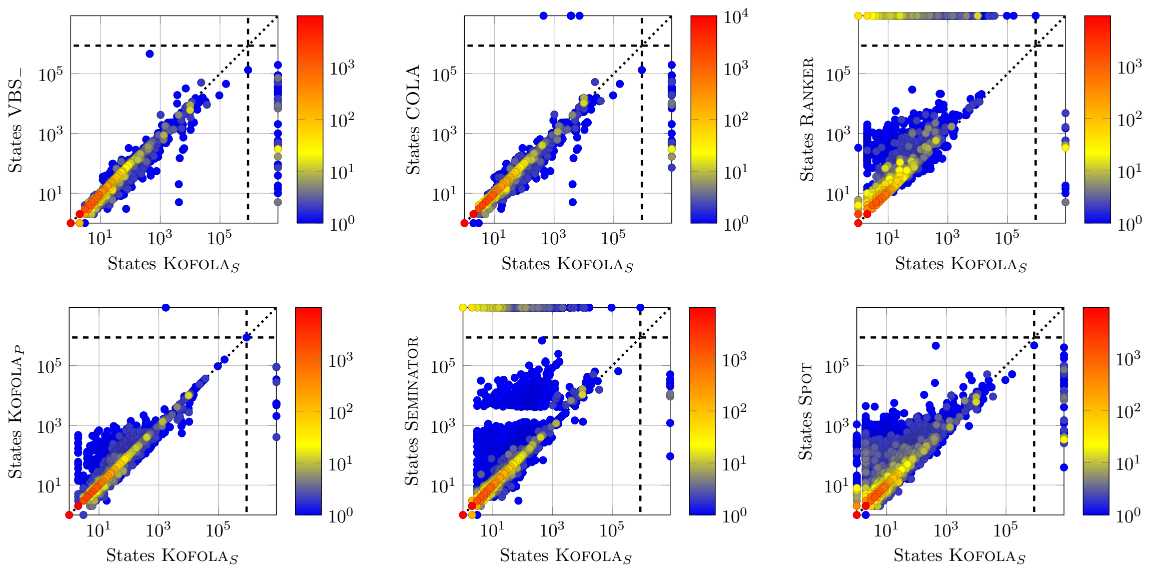


Fig. 2: Scatter plots comparing the numbers of states generated by the tools.

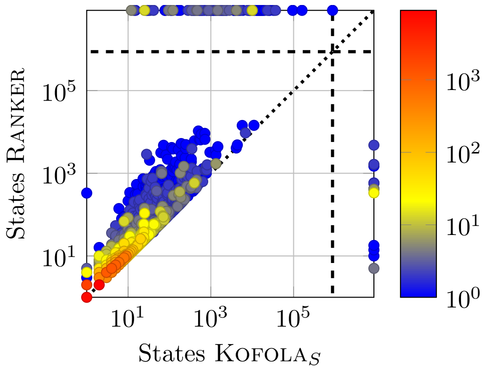
Intel 3.6 GHz i7-4790 CPU. To constrain and collect statistics about the executions of the tools, we used BenchExec [3] and imposed a memory limit of 12 GiB and a timeout of 10 minutes; we used Spot to cross-validate the equivalence of the automata generated by the different tools. An artifact reproducing our experiments is available as [26].

As our data set, we used 39,837 BAs from the automata-benchmarks repository [36] (used before by, e.g., [29,28,37]), which contains BAs from the following sources: (i) randomly generated BAs used in [52] (21,876 BAs), (ii) BAs obtained from LTL formulae from the literature and randomly generated LTL formulae [5] (3,442 BAs), (iii) BAs obtained from Ultimate Automizer [11] (915 BAs), (iv) BAs obtained from the solver for first-order logic over Sturmian words Pecan [31] (13,216 BAs), (v) BAs obtained from an S1S solver [23] (370 BAs), and (vi) BAs from LTL to SDBA translation [49] (18 BAs). From these BAs, 23,850 are deterministic, 6,147 are SDBAs (but not deterministic), 4,105 are elevator (but not SDBAs), and 5,735 are the rest.

In Table 1 we present an overview of the outcomes. Despite being a prototype, Kofola can already complement a large portion of the input automata, with very few cases that can be complemented successfully only by Spot or COLA. Regarding the mean number of states, Kofola$_S$ has the **least mean value** from all tools (except Ranker, which, however, had 1,000 unsolved cases) Moreover, Kofola **significantly decreased the mean number of states** when included into the VBS: from 96 to 78! We consider this to be a strong validation of the usefulness of our approach. Regarding the runtime, both versions of Kofola are rather similar; Kofola is just slightly slower than Spot and COLA but much faster than both Ranker and Seminator (cf. [27]).

In Fig. 2 we present a comparison of the number of states generated by Kofola$_S$ and other tools; we omit VBS$_+$ since the corresponding plot can be derived from the one for VBS$_-$ (since Ranker and Seminator only output BAs, we compare the sizes of outputs transformed into BAs for all tools to be fair). In the plots, the number of benchmarks represented by each mark is given by its colour; a mark above the diagonal means that Kofola$_S$ generated a BA smaller than the other tool while a mark on the top border means that the other tool failed while Kofola$_S$ succeeded, and symmetrically for the bottom part and the right-hand border. Dashed lines represent the maximum number of states generated by one of the tools in the plot, axes are logarithmic.

From the results, Kofola$_S$ clearly dominates state-of-the-art tools that are not based on SCC decomposition (Ranker, Spot, Seminator). The outputs are quite comparable to COLA, which also uses SCC decomposition and can be seen as an instantiation of our framework. This supports our intuition that working on the single SCCs helps in reducing the size of the final automaton, confirming the validity of our modular mix-and-match Büchi complementation approach. Lastly, in the figure in the right we compare our algorithm for elevator automata with the one in Ranker (the only other tool with a dedicated algorithm for this subclass). Our new algorithm clearly dominates the one in Ranker.



8 Related Work

To the best of our knowledge, we provide the *first general framework* where one can plug-in different BA complementation algorithms while taking advantage of the specific structure of SCCs. We will discuss the difference between our work and the literature.

The breakpoint construction [42] was designed to complement BAs with only IWCs, while our construction treats it as a partial complementation procedure for IWCs and differs in the need to handle incoming states from other partition blocks. The NCSB family of algorithms [6,11,5,28] for SDBAs do not work when there are nondeterministic jumps between DACs; they can, however, be adapted as partial procedures for complementing DACs in our framework, cf. Sec. 4.2. In [29], a deelevation-based procedure is applied to elevator automata to obtain BAs with a fixed maximum rank of 3, for which a rank-based construction produces a result of the size in $O(16^n)$. In our work, we exploit the structure of the SCCs much more to obtain an exponentially better upper bound of $O(4^n)$ (the same as for SDBAs). The upper bound $O(4^n)$ for complementing unambiguous BAs was established in [39], which is orthogonal to our work, but seems to be possible to incorporate into our framework in the future.

There is a huge body of work on complementation of general BAs [8,50,7,34,21,22,10,24,29,48,2,46,43,45,5,52,32,53,19,20]; all of them work on the whole graph structure of the input BAs. Our framework is general enough to allow including all of them as partial complementation procedures for NACs. On the contrary, our framework does not directly allow (at least in the synchronous strategy) to use algorithms that *do not* work on the structure of the input BA, such as the learning-based complementation algorithm from [38]. The recent determinization algorithm from [37], which serves as our inspiration, also handles SCCs separately (it can actually be seen as an instantiation of our framework). Our current algorithm is, however, more flexible, allowing to mix-and-match various constructions, keep SCCs separate or merge them into partition blocks, and allows to obtain the complexity $O(4^n)$, while [37] only allowed $O(n!)$ (which is tight since SDBA determinization is in $\Omega(n!)$ [17,40]).

Regarding the tool Spot [15,16], it should not be perceived as a single complementation algorithm. Instead, Spot should be seen as a highly engineered platform utilizing breakpoint construction for inherently weak BAs, NCSB [6,11] for SDBAs, and determinization-based complementation [46,43,45] for general BAs, while using many other heuristics along the way. Seminator uses semi-determinization [14,4,5] to make sure the input is an SDBA and then uses NCSB [6,11] to compute the complement.

9 Conclusion and Future Work

We have proposed a general framework for BA complementation where one can plug-in different partial complementation procedures for SCCs by taking advantage of their specific structure. Our framework not only obtains an exponentially better upper bound for elevator automata, but also complements existing approaches well. As shown by the experimental results (especially for the VBS), our framework significantly improves the current portfolio of complementation algorithms.

We believe that our framework is an ideal testbed for experimenting with different BA complementation algorithms, e.g., for the following two reasons: (i) One can develop an efficient complementation algorithm that only works for a quite restricted sub-class of BAs (such as the algorithm for initial deterministic SCCs that we showed in Sec. 5.1) and the framework can leverage it for complementation of all BAs that contain such a substructure. (ii) When one tries to improve a general complementation algorithm, they can focus on complementation of the structurally hard SCCs (mainly the nondeterministic accepting SCCs) and do not need to look for heuristics that would improve the algorithm if there were some easier substructure present in the input BA (as was done, e.g., in [29]). From how the framework is defined, it immediately offers opportunities for being used for on-the-fly BA *language inclusion* testing, leveraging the partial complementation procedures present. Finally, we believe that the framework also enables new directions for future research by developing smart ways, probably based on machine learning, of selecting which partial complementation procedure should be used for which SCC, based on their features. In future, we want to incorporate other algorithms for complementation of NACs, and identify properties of SCCs that allow to use more efficient algorithms (such as unambiguous NACs [39]). Moreover, it seems that generalizing the Delayed optimization from [24] on the top-level algorithm could also help reduce the state space.

Acknowledgements. We thank the reviewers for their useful remarks that helped us improve the quality of the paper and Alexandre Duret-Lutz for sharing a Ti*k*Z package for beautiful automata. This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (grant no. XDA0320000); the National Natural Science Foundation of China (grants no. 62102407 and 61836005); the CAS Project for Young Scientists in Basic Research (grant no. YSBR-040); the Engineering and Physical Sciences Research Council (grant no. EP/X021513/1); the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme; the Czech Science Foundation project GA23-07565S; and the FIT BUT internal project FIT-S-23-8151.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant no. 101008233.

Data Availability Statement. An environment with the tools and data used for the experimental evaluation in the current study is available in the following Zenodo repository: https://doi.org/10.5281/zenodo.7505210.

References

1. Abdulla, P.A., Chen, Y., Holík, L., Vojnar, T.: Mediating for reduction (on minimizing alternating büchi automata). Theor. Comput. Sci. **552**, 26–43 (2014). https://doi.org/10.1016/j.tcs.2014.08.003, https://doi.org/10.1016/j.tcs.2014.08.003
2. Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 46–55. ACM (2018). https://doi.org/10.1145/3209108.3209138, https://doi.org/10.1145/3209108.3209138
3. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y, https://doi.org/10.1007/s10009-017-0469-y
4. Blahoudek, F., Duret-Lutz, A., Klokocka, M., Kretínský, M., Strejcek, J.: Seminator: A tool for semi-determinization of omega-automata. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 356–367. EasyChair (2017). https://doi.org/10.29007/k5nl, https://doi.org/10.29007/k5nl
5. Blahoudek, F., Duret-Lutz, A., Strejcek, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_2, https://doi.org/10.1007/978-3-030-53291-8_2
6. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.: Complementing semi-deterministic Büchi automata. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 770–787. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_49, https://doi.org/10.1007/978-3-662-49674-9_49
7. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Birkedal, L. (ed.) Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7213, pp. 150–164. Springer (2012). https://doi.org/10.1007/978-3-642-28729-9_10, https://doi.org/10.1007/978-3-642-28729-9_10
8. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Mac Lane, S., Siefkes, D. (eds.) The Collected Works of J. Richard Büchi, pp. 425–435. Springer (1990). https://doi.org/10.1007/978-1-4613-8928-6_23, https://doi.org/10.1007/978-1-4613-8928-6_23
9. Bustan, D., Grumberg, O.: Simulation-based Minimization. ACM Transactions on Computational Logic **4**(2), 181–206 (2003)
10. Chen, Y., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Lin, A.W. (ed.) Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 447–467. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_23, https://doi.org/10.1007/978-3-030-34175-6_23

11. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). https://doi.org/10.1145/3192366.3192405, https://doi.org/10.1145/3192366.3192405
12. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15, https://doi.org/10.1007/978-3-642-54792-8_15
13. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019, https://doi.org/10.23638/LMCS-15(1:12)2019
14. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 338–345. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21950, https://doi.org/10.1109/SFCS.1988.21950
15. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 122–129 (2016). https://doi.org/10.1007/978-3-319-46520-3_8, https://doi.org/10.1007/978-3-319-46520-3_8
16. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., 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 - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9, https://doi.org/10.1007/978-3-031-13188-2_9
17. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 426–442 (2017). https://doi.org/10.1007/978-3-662-54577-5_25, https://doi.org/10.1007/978-3-662-54577-5_25
18. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM J. Comput. **34**(5), 1159–1175 (2005). https://doi.org/10.1137/S0097539703420675, https://doi.org/10.1137/S0097539703420675
19. Fogarty, S., Kupferman, O., Vardi, M.Y., Wilke, T.: Profile trees for Büchi word automata, with application to determinization. Inf. Comput. **245**, 136–151 (2015). https://doi.org/10.1016/j.ic.2014.12.021, https://doi.org/10.1016/j.ic.2014.12.021
20. Fogarty, S., Kupferman, O., Wilke, T., Vardi, M.Y.: Unifying Büchi complementation constructions. Log. Methods Comput. Sci. **9**(1) (2013). https://doi.org/10.2168/LMCS-9(1:13)2013, https://doi.org/10.2168/LMCS-9(1:13)2013

21. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. Int. J. Found. Comput. Sci. **17**(4), 851–868 (2006). https://doi.org/10.1142/S0129054106004145, https://doi.org/10.1142/S0129054106004145
22. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2860, pp. 96–110. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_10, https://doi.org/10.1007/978-3-540-39724-3_10
23. Havlena, V., Lengál, O., Smahlíková, B.: Deciding S1S: down the rabbit hole and through the looking glass. In: Echihabi, K., Meyer, R. (eds.) Networked Systems - 9th International Conference, NETYS 2021, Virtual Event, May 19-21, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12754, pp. 215–222. Springer (2021). https://doi.org/10.1007/978-3-030-91014-3_15, https://doi.org/10.1007/978-3-030-91014-3_15
24. Havlena, V., Lengál, O.: Reducing (to) the ranks: Efficient rank-based Büchi automata complementation. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 2:1–2:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.2, https://doi.org/10.4230/LIPIcs.CONCUR.2021.2
25. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Kofola (2022), https://github.com/VeriFIT/kofola
26. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Artifact for the TACAS'23 paper "Modular Mix-and-Match Complementation of Büchi Automata" (Jan 2023). https://doi.org/10.5281/zenodo.7505210, https://doi.org/10.5281/zenodo.7505210
27. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Modular mix-and-match complementation of Büchi automata (technical report). CoRR **abs/2301.01890** (2023). https://doi.org/10.48550/arXiv.2301.01890, https://doi.org/10.48550/arXiv.2301.01890
28. Havlena, V., Lengál, O., Šmahlíková, B.: Complementing Büchi automata with Ranker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 188–201. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_10, https://doi.org/10.1007/978-3-031-13188-2_10
29. Havlena, V., Lengál, O., Šmahlíková, B.: Sky is not the limit: Tighter rank bounds for elevator automata in Büchi automata complementation. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 118–136. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_7, https://doi.org/10.1007/978-3-030-99527-0_7
30. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) 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. 797–813. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_53, https://doi.org/10.1007/978-3-319-08867-9_53
31. Hieronymi, P., Ma, D., Oei, R., Schaeffer, L., Schulz, C., Shallit, J.O.: Decidability for Sturmian words. In: Manea, F., Simpson, A. (eds.) 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference). LIPIcs, vol. 216, pp. 24:1–24:23. Schloss Dagstuhl - Leibniz-Zentrum für

Informatik (2022). https://doi.org/10.4230/LIPIcs.CSL.2022.24, https://doi.org/10.4230/LIPIcs.CSL.2022.24

32. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games. Lecture Notes in Computer Science, vol. 5125, pp. 724–735. Springer (2008). https://doi.org/10.1007/978-3-540-70575-8_59, https://doi.org/10.1007/978-3-540-70575-8_59
33. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Liu, Z., Ravn, A.P. (eds.) Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5799, pp. 228–243. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_18, https://doi.org/10.1007/978-3-642-04761-9_18
34. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993, https://doi.org/10.1145/377978.377993
35. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. **35**(1), 59–71 (1987). https://doi.org/10.1016/0022-0000(87)90036-5, https://doi.org/10.1016/0022-0000(87)90036-5
36. Lengál, O.: Automata benchmarks (2022), https://github.com/ondrik/automata-benchmarks
37. Li, Y., Turrini, A., Feng, W., Vardi, M.Y., Zhang, L.: Divide-and-conquer determinization of Büchi automata based on SCC decomposition. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 152–173. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_8, https://doi.org/10.1007/978-3-031-13188-2_8
38. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10747, pp. 313–335. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_15, https://doi.org/10.1007/978-3-319-73721-8_15
39. Li, Y., Vardi, M.Y., Zhang, L.: On the power of unambiguity in Büchi complementation. In: Raskin, J., Bresolin, D. (eds.) Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21-22, 2020. EPTCS, vol. 326, pp. 182–198 (2020). https://doi.org/10.4204/EPTCS.326.12, https://doi.org/10.4204/EPTCS.326.12
40. Löding, C.: Optimal bounds for transformations of omega-automata. In: Rangan, C.P., Raman, V., Ramanujam, R. (eds.) Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1738, pp. 97–109. Springer (1999). https://doi.org/10.1007/3-540-46691-6_8, https://doi.org/10.1007/3-540-46691-6_8
41. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of Büchi automata. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 317–333. Springer

(2019). https://doi.org/10.1007/978-3-030-31784-3_18, https://doi.org/10.1007/978-3-030-31784-3_18

42. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. **32**, 321–330 (1984). https://doi.org/10.1016/0304-3975(84)90049-5, https://doi.org/10.1016/0304-3975(84)90049-5
43. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007, https://doi.org/10.2168/LMCS-3(3:5)2007
44. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32, https://doi.org/10.1109/SFCS.1977.32
45. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744, https://doi.org/10.3233/FI-2012-744
46. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948, https://doi.org/10.1109/SFCS.1988.21948
47. Safra, S., Vardi, M.Y.: On omega-automata and temporal logic (preliminary report). In: Johnson, D.S. (ed.) Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA. pp. 127–137. ACM (1989). https://doi.org/10.1145/73007.73019, https://doi.org/10.1145/73007.73019
48. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854, https://doi.org/10.4230/LIPIcs.STACS.2009.1854
49. Sickert, S., Esparza, J., Jaax, S., Kretínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 312–332. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_17, https://doi.org/10.1007/978-3-319-41540-6_17
50. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987). https://doi.org/10.1016/0304-3975(87)90008-9, https://doi.org/10.1016/0304-3975(87)90008-9
51. The SV-COMP Community: International competition on software verification (2022), https://sv-comp.sosy-lab.org/
52. Tsai, M., Fogarty, S., Vardi, M.Y., Tsay, Y.: State of Büchi complementation. Log. Methods Comput. Sci. **10**(4) (2014). https://doi.org/10.2168/LMCS-10(4:13)2014, https://doi.org/10.2168/LMCS-10(4:13)2014
53. Vardi, M.Y., Wilke, T.: Automata: from logics to algorithms. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata: History and Perspectives. Texts in Logic and Games, vol. 2, pp. 629–736. Amsterdam University Press (2008)
54. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986. pp. 332–344. IEEE Computer Society (1986)

55. Yan, Q.: Lower bounds for complementation of omega-automata via the full automata technique. Log. Methods Comput. Sci. **4**(1) (2008). https://doi.org/10.2168/LMCS-4(1:5)2008, https://doi.org/10.2168/LMCS-4(1:5)2008

Advanced Automata-Based Algorithms for Program Termination Checking

Yu-Fang Chen
Academia Sinica, Taiwan
National Taipei University, Taiwan
yfc@iis.sinica.edu.tw

Matthias Heizmann
University of Freiburg, Germany
heizmann@informatik.uni-freiburg.de

Ondřej Lengál
FIT, Brno University of Technology
IT4Innovations Centre of Excellence
Czech Republic, lengal@fit.vutbr.cz

Yong Li
State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China, liyong@ios.ac.cn

Ming-Hsien Tsai
Academia Sinica, Taiwan
mhtsai208@gmail.com

Andrea Turrini
State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences
China
turrini@ios.ac.cn

Lijun Zhang
State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China, zhanglj@ios.ac.cn

Abstract

In 2014, Heizmann *et al.* proposed a novel framework for program termination analysis. The analysis starts with a termination proof of a sample path. The path is generalized to a Büchi automaton (BA) whose language (by construction) represents a set of terminating paths. All these paths can be safely removed from the program. The removal of paths is done using automata difference, implemented via BA complementation and intersection. The analysis constructs in this way a set of BAs that jointly "cover" the behavior of the program, thus proving its termination. An implementation of the approach in Ultimate Automizer won the 1st place in the Termination category of SV-Comp 2017.

In this paper, we exploit advanced automata-based algorithms and propose several non-trivial improvements of the framework. To alleviate the complementation computation for BAs—one of the most expensive operations in the framework—, we propose a multi-stage generalization construction. We start with generalizations producing subclasses of BAs (such as deterministic BAs) for which efficient complementation algorithms are known, and proceed to more general classes only if necessary. Particularly, we focus on the quite expressive subclass of semideterministic BAs and provide an improved complementation algorithm for this class. Our experimental evaluation shows that the proposed approach significantly improves the power of termination checking within the Ultimate Automizer framework.

CCS Concepts **• Theory of computation → Automata over infinite objects**; **• Software and its engineering → Formal software verification**;

Keywords Program Termination, Büchi Automata Complementation and Language Difference

ACM Reference Format:
Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced Automata-Based Algorithms for Program Termination Checking. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192405

1 Introduction

Termination analysis of programs is a challenging area of formal verification, which has attracted the interest of many researchers approaching the problem from different angles [4, 13, 17–19, 27, 28, 31, 33, 34, 36, 38, 42, 45–48, 51, 52]. All approaches need to deal with the following challenge: when a program contains loops with branching or nesting, how to

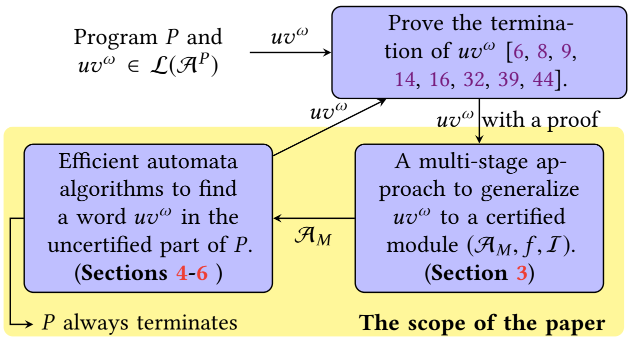

Program P and $uv^\omega \in \mathcal{L}(\mathcal{A}^P)$
uv^ω
Prove the termination of uv^ω [6, 8, 9, 14, 16, 32, 39, 44].
uv^ω
uv^ω with a proof
Efficient automata algorithms to find a word uv^ω in the uncertified part of P. (Sections 4-6)
$\mathcal{A}_M$
A multi-stage approach to generalize uv^ω to a certified module $(\mathcal{A}_M, f, \mathcal{I})$. (Section 3)
P always terminates
The scope of the paper

Figure 1. The flow of the automata-based termination analysis and the scope of the paper

devise a termination argument that holds for any possible interleaving of different paths through the loop body?

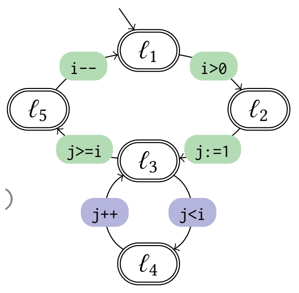
Due to the difficulty of solving the general problem, many researchers have focused on its simplified version that addresses only *lasso-shaped* programs, i.e., programs where the control flow consists of a stem followed by a simple loop without any branching. Proving termination of this class of programs can be done rather efficiently [6, 8, 9, 14, 16, 32, 39, 44].

The approach of Heizmann *et al.* [33] leverages those results and proposes a modular construction of termination proofs for a general program P from termination proofs of lasso-shaped programs obtained from its concrete paths. On a high level, the approach repeatedly performs the following sequence of operations (see Figure 1). First, it samples a path $\tau = uv^\omega$ from the possible behaviors of P and attempts to prove its termination using an off-the-shelf approach. Second, it generalizes τ into a (potentially infinite) set of paths $\mathcal{M}$, called a *certified module*, that all share the same termination proof with τ. Finally, it checks whether the behavior of P contains a path τ' not covered by any certified module generated so far and, if so, the procedure is restarted. This sequence is repeated until either a non-terminating path is found or all behaviors of P are covered by the modules.

Let us follow with an informal description of the procedure on the example program P^{sort} in Figure 2a. Figure 2b shows the control flow graph (CFG) of P^{sort} as a Büchi automaton (BA) $\mathcal{A}^{P^{\mathsf{sort}}}$. The alphabet of $\mathcal{A}^{P^{\mathsf{sort}}}$ is the set of all statements occurring in P^{sort} and each state of $\mathcal{A}^{P^{\mathsf{sort}}}$ is a location of P^{sort}. All states of $\mathcal{A}^{P^{\mathsf{sort}}}$ are accepting so every infinite sequence of statements of the program corresponds to an infinite word in the language $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}})$. The task is to decompose $\mathcal{A}^{P^{\mathsf{sort}}}$ into a finite set of BAs $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, each representing a program with a termination argument, such that $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \cdots \cup \mathcal{L}(\mathcal{A}_n)$, so every path of P^{sort} is represented by a word in $\mathcal{A}^{P^{\mathsf{sort}}}$ and is guaranteed to terminate by an argument for some $\mathcal{A}_i$. More concretely, each BA $\mathcal{A}_i$ is associated with a *ranking function* f_i and a *rank certificate* $\mathcal{I}_i$ mapping each state to a *predicate* over the program variables (cf. Section 3). The triple $\mathcal{M}_i = (\mathcal{A}_i, f_i, \mathcal{I}_i)$ is called

```
program sort(int i):
ℓ1: while (i>0):
ℓ2:   int j:=1
ℓ3:   while (j<i):
        // if (a[j]>a[i]):
        //   swap(a[j],a[i])
ℓ4:     j++
ℓ5:   i--
```

(a) Program P^{sort}



(b) The BA $\mathcal{A}^{P^{\mathsf{sort}}}$

Figure 2. An example program and its BA representation

a *certified module*. The construction of the set $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ is step-wise (see Figure 1). First, we find an ultimately periodic word $uv^\omega \in \mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}})$—which is essentially a lasso-shaped program—and use a standard approach to check if it corresponds to a terminating path. In our example, we start with sampling the word uv^ω = `i>0` `j:=1` (`j<i` `j++`)$^\omega$. We can prove termination of the path corresponding to uv^ω by finding, e.g., the ranking function $f_1(\mathtt{i}, \mathtt{j}) = \mathtt{i} - \mathtt{j}$.

In the following, we will denote the outer loop of $\mathcal{A}^{P^{\mathsf{sort}}}$ as Outer = `j>=i` `i--` `i>0` `j:=1` and its inner loop as Inner = `j<i` `j++`. We can observe that f_1 is also a ranking function for the set of paths obtained by generalizing uv^ω into the set of words that correspond to all paths that eventually stay in the inner loop, i.e., words from

$$\mathcal{L}_1 = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner} + \textsc{Outer})^* \cdot \textsc{Inner}^\omega. \quad (1)$$

The language $\mathcal{L}_1$ together with a ranking function f_1 and a rank certificate $\mathcal{I}_1$ can be represented by the certified module $\mathcal{M}_1 = (\mathcal{A}_1, f_1, \mathcal{I}_1)$ where $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}_1$. We proceed by removing all paths covered by $\mathcal{L}_1$ from $\mathcal{A}^{P^{\mathsf{sort}}}$ to know which paths still need to be examined. The removal can be performed by executing a *BA difference algorithm*, followed by checking language emptiness (potentially finding a new counterexample uv^ω on failure). In our example, the difference corresponds to the (non-empty) language

$$\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1}) = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner}^*\textsc{Outer})^\omega \quad (2)$$

represented by $\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1}$. Suppose the next sampling gives us uv^ω = `i>0` `j:=1` $\cdot$ Outer$^\omega$, for which, e.g., the ranking function $f_2(\mathtt{i}, \mathtt{j}) = \mathtt{i}$ is applicable. Note that f_2 is also a valid ranking function for all paths taking the outer while loop infinitely often, i.e., all paths corresponding to words from

$$\mathcal{L}_2 = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner}^*\textsc{Outer})^\omega. \quad (3)$$

We represent these paths by the certified module $\mathcal{M}_2 = (\mathcal{A}_2, f_2, \mathcal{I}_2)$ where $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}_2$. After removing the words from $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1})$, we, finally, obtain the BA $\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1,\mathcal{A}_2}$, whose language is empty. This means that the modules $\mathcal{M}_1$ and $\mathcal{M}_2$ cover all possible paths of the program P^{sort} and, because each of them comes with a termination argument, we can conclude that all paths of P^{sort} are guaranteed to terminate.

Note that the above procedure performs extensively computation of *language difference* of a pair of BAs. The computation of difference involves computing the *complement* of a BA, one of the most difficult operations in automata theory—it is known that complementing a BA with n states has the lower-bound space complexity $2^{O(n \log n)}$ [40].

In this paper, we exploit advanced automata-based algorithms and propose several non-trivial improvements of the framework. Our main contributions are the following:

Contribution 1: We devise a *multi-stage generalization* approach, which tries to avoid the costly complementation of general BAs by considering several subclasses of BAs with cheaper complementation operations. For every terminating trace represented as a word uv^ω, we consider the following subclasses: (i) *finite-trace BAs* (BAs accepting the language $w\Sigma^\omega$ for a word $w \in \Sigma^*$), (ii) *deterministic BAs* (DBAs), (iii) *semideterministic BAs* (SDBAs; BAs where, intuitively, all strongly connected components (SCCs) containing an accepting state are deterministic), and, finally, (iv) general *BAs*. These subclasses indeed have more efficient complementation procedures: the complementation of finite-trace BAs needs only $O(1)$ space, DBAs can be complemented in $O(n)$ space [35], and complementing SDBAs requires only $2^{O(n)}$ space [12]. The details of the multi-stage approach are presented in Section 3. Our observation from running the multi-stage approach is that general BAs are needed only rarely—in the vast majority of cases, SDBAs are expressive enough.

Contribution 2: In our multi-stage approach shown above, the computation of the difference automaton for a BA and an SDBA is one of the most expensive operations in the loop of automata-based termination analysis. Our second contribution is an efficient algorithm for computing the language difference of a BA and an SDBA. The difference algorithm performs on-the-fly intersection and complement, on the top level using the (as far as we know currently the most efficient) SCC-based BA emptiness checking algorithm of [26]. The details of the algorithm are given in Section 4.

Contribution 3: Our third contribution is the improvement of the efficiency of state-of-the-art algorithms manipulating SDBAs. We, in particular, provide several heuristics of the SDBA complementation procedure of Blahoudek *et al.* from [12]. The main ideas of the heuristics are the following: (i) *lazy* construction, which delays nondeterministic choices in a way similar to partial order reduction [30, 43, 54] (Section 5), and (ii) *subsumption*-based pruning of states inspired by antichain-based algorithms for testing universality and language inclusion over finite automata [2, 23] (Section 6).

We implemented the proposed solutions in the open source tool Ultimate Automizer and evaluated them on the benchmarks from SV-Comp [1]. Our experimental evaluation (Section 7) shows that the approach we propose in this work has significantly improved the power of termination checking within the Ultimate Automizer framework.

2 Preliminaries

We fix an *alphabet* Σ. A (nondeterministic) *generalized Büchi automaton* (GBA) with k accepting sets is a tuple $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F})$, where Q is a finite set of states, $\delta \colon Q \times \Sigma \to 2^Q$ is a transition function, $Q_I \subseteq Q$ is a set of initial states, and $\mathcal{F} = \{ F_j \subseteq Q \mid j \in \{1, \dots, k\} \}$ is a set of accepting conditions. Unless stated explicitly, we assume that all GBAs are *complete*, i.e., for each $q \in Q$ and $a \in \Sigma$, it holds that $\delta(q, a) \neq \emptyset$. We use $q \xrightarrow{a} p$ to denote that $p \in \delta(q, a)$, and we define $post(q) = \bigcup_{a \in \Sigma} \delta(q, a)$. We lift δ to sets of states in the usual way. We abuse notation and for $q \in Q$ use $\mathcal{F}(q) = \{ j \in \{1, \dots, k\} \mid q \in F_j \}$ to denote the set of accepting conditions that q satisfies. Moreover, we sometimes use $\mathcal{F}$ also to denote the set $\{1, \dots, k\}$.

A *trace* of $\mathcal{A}$ on an infinite word $w = w_0 w_1 \ldots \in \Sigma^\omega$ from a state q_0 is an infinite sequence of transitions $\pi = q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots$ such that for each $i \geq 0$, we have $q_i \xrightarrow{w_i} q_{i+1}$. The trace π is *accepting* iff for each $1 \leq j \leq k$, there are infinitely many i such that $q_i \in F_j$, and is *safe* iff for all $i \geq 0$, $q_i \notin \bigcup_{1 \leq j \leq k} F_j$. A *run* $\rho = q_0 q_1 \ldots$ is the projection of a trace to states. The concept that a run is *accepting* or *safe* is defined analogously. The *language* of a state $q \in Q$ in $\mathcal{A}$ is the set $\mathcal{L}_{\mathcal{A}}(q) = \{ w \in \Sigma^\omega \mid \mathcal{A}$ has an accepting trace on w from $q\}$ (denoted also as $\mathcal{L}(q)$ if $\mathcal{A}$ is obvious). If $\mathcal{L}(q) = \emptyset$, we call q *useless*. The language of the GBA $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \bigcup_{q_i \in Q_I} \mathcal{L}(q_i)$. We use $\subseteq_{\mathcal{L}}$ to denote the relation of language inclusion of states: $p \subseteq_{\mathcal{L}} q \iff \mathcal{L}(p) \subseteq \mathcal{L}(q)$.

A *Büchi automaton* (BA) is a GBA with just $k = 1$ accepting condition, i.e., $\mathcal{F} = \{F\}$. We often denote a BA as (Q, δ, Q_I, F). A *complement* of $\mathcal{A}$ is a BA $\mathcal{A}_C$ that accepts the language $\mathcal{L}(\mathcal{A}_C) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$. $\mathcal{A}$ is a *deterministic BA* (DBA) if $\forall q \in Q, a \in \Sigma : |\delta(q, a)| \leq 1 \land |Q_I| = 1$. Moreover, $\mathcal{A}$ is a *semideterministic BA* (SDBA) if, for each $q_f \in F$, the automaton $\mathcal{A}(q_f)$ is deterministic, where $\mathcal{A}(q_f)$ is obtained from the BA $(Q, \delta, \{q_f\}, F)$ by removing states unreachable from q_f. Intuitively, this means that an SDBA can move nondeterministically until it visits an accepting state; then it can only move deterministically. The set Q can be divided into two disjoint parts Q_1 and Q_2 representing the states in the nondeterministic part and deterministic part, respectively; note that $F \subseteq Q_2$. The transition function $\delta = \delta_1 \cup \delta_t \cup \delta_2$ then consists of the following three disjoint transition functions: $\delta_1 \colon Q_1 \times \Sigma \to 2^{Q_1}$, $\delta_t \colon Q_1 \times \Sigma \to 2^{Q_2}$, and $\delta_2 \colon Q_2 \times \Sigma \to 2^{Q_2}$, where the relation δ_2 is deterministic. To simplify presentation, we impose the following two additional requirements on SDBAs: (1) we require that the entry points of Q_2 are accepting, i.e., $\delta_t(Q_1, a) \subseteq F$ for all $a \in \Sigma$ and (2) we require that $Q_I \cap Q_2 \subseteq F$, i.e., all initial states in Q_2 are accepting. The two requirements guarantee that if a run in an SDBA touches a state in Q_2, it has already touched some accepting state. Any SDBA can be transformed into an equivalent SDBA satisfying the requirements by treating all non-accepting entry or initial states q from Q_2 (i.e., states from either $(\bigcup_{a\in\Sigma} \delta_t(Q_1, a)) \setminus F$ or $(Q_I \cap Q_2) \setminus F$) as follows: (1) we add a new accepting state q', (2) for all transitions entering q from Q_1, we redirect them to q', and (3) we duplicate all outgoing transitions of q to q'. Note that SDBAs recognize the same class of languages as BAs, but can be, in the worst case, exponentially larger.

3 Multi-Stage Generalization of Certified Modules

As mentioned in the introduction, a program P is represented by a BA $\mathcal{A}$. The termination proof of P can be obtained by decomposing $\mathcal{A}$ into several BAs $\mathcal{A}_1, \ldots, \mathcal{A}_n$, whose languages jointly cover $\mathcal{L}(\mathcal{A})$, and then showing that each of them is terminating by means of a certified module [33].

Given a well-ordered set $(W, \prec)$, let ∞ denote a value strictly larger than any other value in W. In the following, we use $\vec{v}$ to denote the vector of program variables of P. A *valuation* Φ is a function assigning a value to each variable from $\vec{v}$. A *statement* is a command appearing in the program, such as an assignment or the guard of a `while` loop. The alphabet Σ is the set of statements appearing in P. Each statement is associated with a binary relation over valuations representing the effect of the statement; for instance, the relation associated with the statement `i>0` contains the pairs (Φ, Φ) where $\Phi(\mathtt{i}) \implies \mathtt{i} > 0$. A *Hoare triple* is a triple $\{\psi\}\ stmt\ \{\psi'\}$ where $stmt$ is a statement and ψ, ψ' are predicates over program variables; a Hoare triple is valid if, for each pair of valuations (Φ, Φ') in the relation associated with $stmt$, if Φ satisfies ψ, then Φ' satisfies ψ'.

Definition 3.1 (cf. [33, Definition 3]). Given a BA $\mathcal{A}_\mathcal{M} = (Q, \delta, \{q_i\}, \{q_f\})$ and a *ranking function* $f_\mathcal{M}$ from valuations into a well-ordered set $(W, \prec)$, we call a mapping $\mathcal{I}_\mathcal{M}$ from states to predicates over program variables a *rank certificate* for $f_\mathcal{M}$ and $\mathcal{A}_\mathcal{M}$ if the following properties hold:

- The initial state q_i is mapped by $\mathcal{I}_\mathcal{M}$ to the predicate where the auxiliary variable `oldrnk` has the value ∞, i.e., $\mathcal{I}_\mathcal{M}(q_i) \iff \mathtt{oldrnk} = \infty$.
- The accepting state q_f is mapped by $\mathcal{I}_\mathcal{M}$ to a predicate in which the value of f over the program variables is strictly smaller than the value of the variable `oldrnk`, i.e., $\mathcal{I}_\mathcal{M}(q_f) \implies f_\mathcal{M}(\vec{v}) \prec \mathtt{oldrnk}$.
- Each outgoing transition $q \xrightarrow{stmt} q'$ from a state $q \neq q_f$ corresponds to a valid Hoare triple, i.e., the triple $\{\mathcal{I}_\mathcal{M}(q)\}\ stmt\ \{\mathcal{I}_\mathcal{M}(q')\}$ is valid.
- Each outgoing edge $q_f \xrightarrow{stmt} q'$ from the accepting state q_f corresponds to a valid Hoare triple if we insert an additional assignment statement that updates `oldrnk` with the value of the ranking function, i.e., $\{\mathcal{I}_\mathcal{M}(q_f)\}$ `oldrnk:=` $f_\mathcal{M}(\vec{v})$; $stmt$ $\{\mathcal{I}_\mathcal{M}(q')\}$ is valid.

We call $\mathcal{M} = (\mathcal{A}_\mathcal{M}, f_\mathcal{M}, \mathcal{I}_\mathcal{M})$ a *certified module* and define its language as $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A}_\mathcal{M})$. A certified module represents a set of paths in P that share the same termination argument. That is, for all paths represented by $\mathcal{M}$, the evaluation of the ranking function $f_\mathcal{M}$ strictly decreases on visiting the accepting state q_f.

3.1 The Multi-Stage Approach to Construct $\mathcal{M}$

In this section, we describe our algorithm that generalizes an ultimately periodic word uv^ω accompanied by a termination proof (obtained using an off-the-shelf termination checker) into a certified module (cf. Figure 1).

First, we construct the *initial certified lasso module* $\mathcal{M}_{uv^\omega}$ (cf. Section 3.1.1), which closely resembles the structure of uv^ω. The alphabet Σ of $\mathcal{M}_{uv^\omega}$ (and of its generalizations, see below) consists of all statements occurring in uv^ω. While such a module would work correctly in the later refinement, it is of a very limited practical use. In our experience, it usually covers only a very small fragment of program paths; sometimes it only covers the path corresponding to uv^ω.

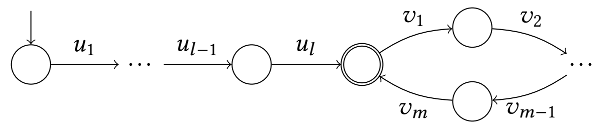
The previous work [33] uses a generalization procedure that uses $\mathcal{M}_{uv^\omega}$ to construct a module $\mathcal{M}_{nondet}$ consisting of a nondeterministic BA (cf. Section 3.1.5). Although $\mathcal{M}_{nondet}$ is usually quite general, the drawback of this solution is the extremely high complexity of complementing a nondeterministic BA, which is performed in the subsequent step. To alleviate this issue, we propose the following multi-stage approach for construction of certified modules.

Our multi-stage approach attempts to use the alphabet and states of $\mathcal{M}_{uv^\omega}$ to construct a certified module that is as easy to complement as possible, while also satisfying the condition that its language contains the word uv^ω (so that when its language is removed from the set of uncertified traces, we are guaranteed to remove at least uv^ω). The construction proceeds in stages, starting with a module that is the easiest to complement, and gradually progresses to modules whose complementation is harder (they exhibit a higher degree of nondeterminism), until it builds a module whose language contains uv^ω. As the last option, we construct $\mathcal{M}_{nondet}$, which is guaranteed to contain uv^ω in its language, but is the hardest to complement.

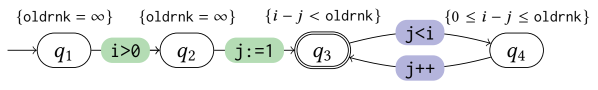
In this work we consider 4 stages: besides stage 0, where the initial certified lasso module $\mathcal{M}_{uv^\omega}$ is built, we have at stage 1 the *finite-trace certified module* construction (cf. Section 3.1.2). The result contains a finite-trace BA whose complementation takes constant time; the generalization is, however, possible only under certain conditions. At stage 2, we build the *deterministic certified module* (cf. Section 3.1.3), which is relatively easy to complement since it is deterministic. If it does not suffice, at stage 3, we create the *semideterministic certified module* (cf. Section 3.1.4), which allows limited nondeterminism. The last construction we consider at stage 4 is the *nondeterministic certified module* $\mathcal{M}_{nondet}$ (cf. Section 3.1.5), which is guaranteed to accept uv^ω but has the highest level of nondeterminism. More intermediate constructions can be added into this multi-stage approach.

3.1.1 Stage 0: Initial Certified Lasso Module

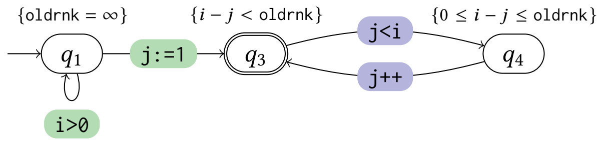
The *initial certified lasso module* $\mathcal{M}_{uv^\omega}$ consists of a BA accepting solely the word uv^ω, a ranking function f, and a rank certificate $\mathcal{I}$ (from a lasso program termination prover). The construction starts with the BA of the form depicted below, for the *stem* $u = u_1 u_2 \dots u_l$ and the *loop* $v = v_1 v_2 \dots v_m$.[1]



For instance, consider again the sorting program P^{sort} and the ω-word uv^ω = `i>0` `j:=1` (`j<i` `j++`)$^\omega$. The corresponding BA and rank certificate $\mathcal{I}^{\mathsf{sort}}$ are depicted below, where each state is annotated with the corresponding predicate given by $\mathcal{I}^{\mathsf{sort}}$ for the ranking function $f^{\mathsf{sort}}(i, j) = i - j$.



Module $\mathcal{M}_{uv^\omega}$ is obtained by generalizing the constructed BA by merging states with the same predicate. Note that for the given word uv^ω, two states can be merged only if they both belong to the stem part u or both belong to the loop part v; a state from the stem part can never be merged with a state from the loop part, since the former implies `oldrnk` $= \infty$ in its predicate while the latter implies its negation. If we merge such states for the BA from above, we obtain the following module accepting all words of the form (`i>0`)* `j:=1` (`j<i` `j++`)$^\omega$, not just uv^ω = `i>0` `j:=1` (`j<i` `j++`)$^\omega$.



We denote the module from the example above as $\mathcal{M}^{\mathsf{sort}}_{uv^\omega}$.

3.1.2 Stage 1: Finite-trace Certified Module

The first module we try to construct is the *finite-trace certified module* $\mathcal{M}_1 = \mathcal{M}_{\mathsf{fin}}$. This module can be constructed when uv^ω corresponds to a path that is infeasible already in the stem part. In such a case, there must be a state q on some path from the initial state q_i to the accepting state q_f s.t. $\mathcal{I}(q)$ is unsatisfiable and for every q' on the shortest path from q_i to q, $\mathcal{I}(q')$ is satisfiable. More concretely, $\mathcal{M}_{\mathsf{fin}}$ can be constructed from $\mathcal{M}_{uv^\omega}$ by (1) removing all states that are not on a path from q_i to q, (2) removing all outgoing transitions of q, (3) adding self-loops $q \xrightarrow{stmt} q$ for all $stmt \in \Sigma$, and (4) setting q as the single new accepting state.

3.1.3 Stage 2: Deterministic Certified Module

If $\mathcal{M}_1$ cannot be constructed, we proceed by building a *deterministic certified module* $\mathcal{M}_2 = \mathcal{M}_{\mathsf{det}}$. The high-level intuitive idea is to construct a DBA using sets of states of $\mathcal{M}_{uv^\omega}$ with transitions that respect the predicates of $\mathcal{M}_{uv^\omega}$ (so that the termination argument for uv^ω correctly extends to the whole language of the module) and are in some sense *maximal*. In particular, the successor of a set of states $\mathfrak{Q}$ of $\mathcal{M}_{uv^\omega}$ over a statement *stmt* is computed as the maximal set of states $\mathfrak{Q}'$ satisfying the following property: the predicate of every state in $\mathfrak{Q}'$ is a logical consequence of the conjunction of the predicates of the states in $\mathfrak{Q}$ and the semantics of *stmt*.

For instance, let us consider an initial certified lasso module $\mathcal{M}_{uv^\omega}$ whose alphabet contains the statement `z:=x+y` and whose states q_{23}, q_{42}, q_{65} are annotated by $\mathcal{I}$ as follows: $\mathcal{I}(q_{23})$ is $x = 23$, $\mathcal{I}(q_{42})$ is $y = 42$, and $\mathcal{I}(q_{65})$ is $z = 65$. Then, regardless of the transitions of $\mathcal{M}_{uv^\omega}$, the successor of $\{q_{23}, q_{42}\}$ over `z:=x+y` is the set $\{q_{23}, q_{42}, q_{65}\}$, since $\mathcal{I}(q_{65})$ is implied by $\mathcal{I}(q_{23}) \wedge \mathcal{I}(q_{42})$ and the relation for `z:=x+y`, which contains all pairs (Φ, Φ') such that $\Phi'(z) \implies z = x+y$ and $\Phi(v) \implies \Phi'(v)$ for $v \neq z$.

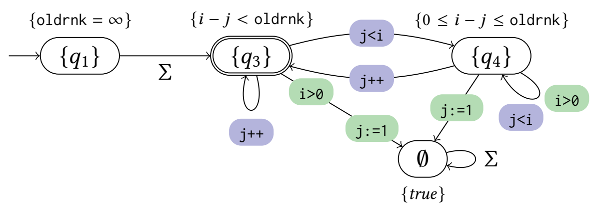
Definition 3.2. Let $\mathcal{M}_{uv^\omega} = (\mathcal{A}, f, \mathcal{I})$ be an initial certified lasso module such that $\mathcal{A} = (Q, \delta, \{q_i\}, \{q_f\})$. The *deterministic certified module* $\mathcal{M}_{\mathsf{det}} = (\mathcal{A}^{\mathsf{det}}, f, \mathcal{I}^{\mathsf{det}})$ with a DBA $\mathcal{A}^{\mathsf{det}} = (Q^{\mathsf{det}}, \delta^{\mathsf{det}}, {Q_I}^{\mathsf{det}}, F^{\mathsf{det}})$ is defined as follows:

- The set of states of $\mathcal{A}^{\mathsf{det}}$ is $Q^{\mathsf{det}} = 2^Q$.
- Let $\delta_\wedge \colon 2^Q \times \Sigma \to 2^Q$ be a function s.t. $\delta_\wedge(\mathfrak{Q}, stmt) = \{q' \in Q \mid \{\bigwedge_{q \in \mathfrak{Q}} \mathcal{I}(q)\}\ stmt'\ \{\mathcal{I}(q')\}$ is a valid Hoare triple$\}$, where $stmt'$ = `oldrnk:=`$f(\vec{v})$; $stmt$ if $q_f \in \mathfrak{Q}$, otherwise $stmt' = stmt$.
  Now, the transition function δ^{det} for a state $\mathfrak{Q} \in Q^{\mathsf{det}}$ and a statement *stmt* is defined as $\delta^{\mathsf{det}}(\mathfrak{Q}, stmt) = \{\mathfrak{Q}'\}$, where $\mathfrak{Q}' = \delta_\wedge(\mathfrak{Q}, stmt)$ if $q_f \notin \delta_\wedge(\mathfrak{Q}, stmt)$, otherwise we omit all non-accepting states whose predicate contains `oldrnk`, i.e., $\mathfrak{Q}' = \delta_\wedge(\mathfrak{Q}, stmt) \setminus \{\, q \in Q \mid q \neq q_f \wedge \texttt{oldrnk} \in \mathrm{var}(\mathcal{I}(q)) \,\}$ where $\mathrm{var}(\mathcal{I}(q))$ denotes all variables occurring in $\mathcal{I}(q)$.
  Note that the statement `oldrnk:=`$f(\vec{v})$ is used only for defining $\delta_\wedge$; it is not in the alphabet of $\mathcal{A}^{\mathsf{det}}$.
- There is a single initial state, i.e., ${Q_I}^{\mathsf{det}} = \{\{q_i\}\}$.
- The set of accepting states F^{det} contains all states $\mathfrak{Q} \in Q^{\mathsf{det}}$ such that $q_f \in \mathfrak{Q}$ or $\bigwedge_{q \in \mathfrak{Q}} \mathcal{I}(q)$ is unsatisfiable.

Moreover, $\mathcal{I}^{\mathsf{det}}$ is such that $\mathcal{I}^{\mathsf{det}} \colon \mathfrak{Q} \to \bigwedge_{q \in \mathfrak{Q}} \mathcal{I}(q)$.

By applying the certified deterministic module construction to $\mathcal{M}^{\mathsf{sort}}_{uv^\omega}$, we obtain the following module:

[1] Note that if $u = \varepsilon$, the construction does not create a certified module (since $q_i = q_f$). As a remedy for this, in such a case we *materialize* v once, i.e., we use the identity $\varepsilon v^\omega = vv^\omega$, and continue in the standard way.



Note that this module $\mathcal{M}^{\mathsf{sort}}_{\mathsf{det}}$, despite accepting a non-empty language, is absolutely useless for the refinement of P^{sort}, since it rejects the word $uv^\omega = \texttt{i>0}\ \texttt{j:=1}\ (\texttt{j<i}\ \texttt{j++})^\omega$ representing the path whose termination has been proved.

While a DBA is easy to use in computing language difference, the certified deterministic module is not always useful, as we can see from the example above (in general, DBAs are known to be less expressive than BAs [3]).

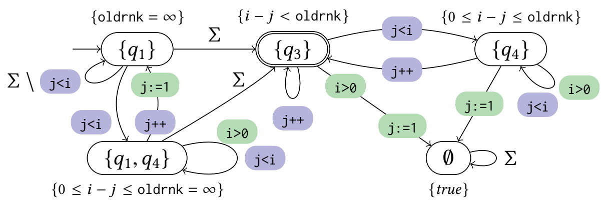
3.1.4 Stage 3: Semideterministic Certified Module

In order to overcome the shortcomings of the deterministic certified module, we now present the *semideterministic certified module* $\mathcal{M}_3 = \mathcal{M}_{\mathsf{semi}}$, which is $\mathcal{M}_{\mathsf{det}}$ enriched with additional transitions. In particular, for a statement *stmt*, each state $\mathfrak{Q}$ that is not reachable from an accepting state s.t. $q_f \in \delta_\wedge(\mathfrak{Q}, stmt)$ (cf. Definition 3.2) has two *stmt*-successors:

- $\delta_\wedge(\mathfrak{Q}, stmt) \setminus \{ q \in Q \mid q \neq q_f \wedge \texttt{oldrnk} \in \mathsf{var}(\mathcal{I}(q)) \}$, the original successor in δ^{det};
- an additional successor $\delta_\wedge(\mathfrak{Q}, stmt) \setminus \{q_f\}$.

Obviously, the resulting automaton is an SDBA that, furthermore, satisfies the requirements from Section 2 (the requirement that any run entering an accepting loop needs to enter via an accepting state—i.e., none of its states contains at the same time a state from the stem and a state from the loop—is guaranteed by the fact that all states in the stem imply $\texttt{oldrnk} = \infty$, while all states in the loop imply $\texttt{oldrnk} < \infty$).

By applying the certified semideterministic module construction to $\mathcal{M}^{\mathsf{sort}}_{uv^\omega}$, we obtain the following module $\mathcal{M}^{\mathsf{sort}}_{\mathsf{semi}}$.



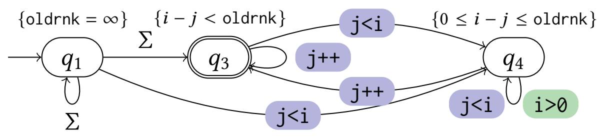
Note that, in contrast to $\mathcal{M}^{\mathsf{sort}}_{\mathsf{det}}$, the module $\mathcal{M}^{\mathsf{sort}}_{\mathsf{semi}}$ already accepts the word $uv^\omega = \texttt{i>0}\ \texttt{j:=1}\ (\texttt{j<i}\ \texttt{j++})^\omega$.

Note that although the construction of the semideterministic module can theoretically produce an exponential-sized automaton, we rarely experienced this case in our evaluation.

3.1.5 Stage 4: Nondeterministic Certified Module

The *nondeterministic certified module* $\mathcal{M}_4 = \mathcal{M}_{nondet}$ is the most powerful generalization we considered. It is obtained by adding every possible transition between each pair of states to $\mathcal{M}_{uv^\omega}$, as long as the rank certificate is still correct.

For instance, the above lasso module $\mathcal{M}^{\mathsf{sort}}_{uv^\omega}$ becomes the following nondeterministic module.



While usually accepting significantly more words than $\mathcal{M}_{uv^\omega}$, the use of $\mathcal{M}_{nondet}$ in the refinement can pose practical problems, caused by its high level of nondeterminism.

Although $\mathcal{M}_{nondet}$ is always guaranteed to accept uv^ω (since it contains all transitions of $\mathcal{M}_{uv^\omega}$), its use in the overall termination procedure is expensive, because algorithms for complementing BAs have a prohibitive complexity. Based on our experiments, constructing $\mathcal{M}_{nondet}$ is seldom necessary, and in many cases, $\mathcal{M}_{\mathsf{semi}}$ is sufficient (in the worst case) for a successful generalization of a program path. As also observed in our experiments, computing the difference of a GBA representing program paths and a module can dominate the overall execution time, so constructing modules that are easier to complement is crucial. In the following sections, we provide efficient algorithms for computing the difference of a GBA and a BA (Section 4) and for complementing an SDBA (Sections 5 and 6) that serve as an enabling technology of the whole termination checking procedure.

4 Building Difference of a GBA and a BA

In this section, we introduce an algorithm that, given a GBA $\mathcal{A}$ (in our setting representing program paths whose termination has not yet been established) and, in general, a BA $\mathcal{B}$ (which represents the program paths whose termination we have just proved), constructs a GBA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$. We present the algorithm and its optimizations in several steps. Note that we use GBAs since they are usually smaller than their equivalent BA counterparts and have a more efficient language intersection operation.

From a high-level view, our algorithm can be seen as an optimization of a naïve algorithm that first builds the complement of $\mathcal{B}$, further denoted as $\overline{\mathcal{B}}$, then constructs a GBA $\mathcal{A}_I$ accepting the intersection $\mathcal{L}(\mathcal{A}_I) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}})$ and, finally, removes useless states from $\mathcal{A}_I$ (yielding an empty automaton in the case $\mathcal{L}(\mathcal{A}_I) = \emptyset$). Recall that a state q is useless iff $\mathcal{L}_{\mathcal{A}_I}(q) = \emptyset$, otherwise, q is *useful*. Our optimizations that make the algorithm usable in practice are the following.

1. $\overline{\mathcal{B}}$ is constructed *on the fly* when constructing $\mathcal{A}_I$, i.e., only those states of $\overline{\mathcal{B}}$ that occur in some product state of $\mathcal{A}_I = \mathcal{A} \cap \overline{\mathcal{B}}$ are constructed (note that intersection of GBAs produces a GBA whose structure corresponds to finite automaton-like product construction).

Algorithm 1: Removing useless states from a GBA

```
Input  :GBA $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F})$
Output :GBA $\mathcal{A}' = (Q', \delta', Q_I', \mathcal{F}')$ s.t. $\forall q \in Q' : \mathcal{L}(q) \neq \emptyset$
Global :$Q' \leftarrow \emptyset$, emp $\leftarrow \emptyset$, SCCs $\leftarrow \emptyset$, act $\leftarrow \emptyset$, cnt $\leftarrow 0$
1  Function remove_useless($\mathcal{A}$):
2      foreach $q_I \in I$ do
3          if $q_I \notin Q' \cup$ emp then    // $q_I \notin Q' \cup \lceil$emp$\rceil$
4              construct($q_I$);
5      return $\mathcal{A}' = (Q', \delta \cap (Q' \times \Sigma \times 2^{Q'}), I \cap Q', \mathcal{F}_{|Q'})$;
6  Function construct($s$):
7      ++cnt; $s$.dfsnum $\leftarrow$ cnt; is_nemp $\leftarrow$ false;
8      SCCs.push(($s, \mathcal{F}(s)$)); act.push($s$);
9      foreach $t \in post(s)$ do
10         if $t \in Q'$ then is_nemp $\leftarrow$ true;
11         else if $t \in$ emp then continue; // $t \in \lceil$emp$\rceil$
12         else if $t \notin$ act then
13             is_nemp $\leftarrow$ construct($t$) $\vee$ is_nemp;
14         else
15             B $\leftarrow \emptyset$;
16             do
17                 $(u, C) \leftarrow$ SCCs.pop(); $B \leftarrow B \cup C$;
18                 if $B = \mathcal{F}$ then is_nemp $\leftarrow$ true ;
19             while $u$.dfsnum $> t$.dfsnum;
20             SCCs.push(($u, B$));
21     if SCCs.top() = ($s$, _) then
22         SCCs.pop();
23         do
24             $u \leftarrow$ act.pop();
25             if is_nemp then $Q'$.add($u$);
26             else emp.add($u$);
27         while $u \neq s$;
28     return is_nemp;
```

2. We remove useless states from $\mathcal{A}_I$ using a modification of the state-of-the-art SCC-based algorithm for testing emptiness of the language of a GBA by Gaiser & Schwoon [26], which refines the algorithm of Couvreur [22] (Section 4.1).
3. When $\mathcal{B}$ is an SDBA, we optimize the construction of $\overline{\mathcal{B}}$ from [12] by *delaying nondeterministic choices* as long as possible, thus significantly reducing the number of generated states (Section 5).
4. We *prune the search* from Point 2 by using an antichain-like [23] subsumption on the states of $\mathcal{A}_I$ (Section 6).

4.1 Removing Useless States in a GBA

Algorithm 1 is a modification of the algorithm for checking emptiness of a GBA $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F} = \{F_1, \ldots, F_k\})$ proposed by Gaiser & Schwoon [26] (GS for short), which is based on finding a reachable strongly connected component (SCC) that contains at least one state from every set F_j. Our modification not only tests the emptiness of $\mathcal{L}(\mathcal{A})$, but also efficiently constructs a copy $\mathcal{A}'$ of $\mathcal{A}$ without any useless state (and, therefore, if $\mathcal{L}(\mathcal{A}) = \emptyset$, then $\mathcal{A}'$ is empty).

Similarly to GS, Algorithm 1 uses two stacks, `SCCs` and `act`, to keep track of the possible entry states of SCCs and the *active* states, which may be constituting the SCCs. Our algorithm uses additional data structures, namely the pair of sets `Q'` and `emp`, which are used to store all states that have been proved to be useful or useless. The algorithm starts in the function `remove_useless` and traverses the reachable states of $\mathcal{A}$ in a depth first search manner. Each state has the data field `dfsnum`, which is used to record the relative order of the visit of the states, i.e., if t.`dfsnum` $>$ s.`dfsnum`, then s has been visited before t. Therefore, if such a t can reach the said s and, at the same time, s is in `act`, this means that $\mathcal{A}$ contains an SCC that includes both s and t. From all states forming an SCC, the one with the lowest value of `dfsnum` is the SCC's entry point. The stack `SCCs` also assigns each possible SCC entry point q_e the set of accepting conditions from $\mathcal{F}$ that q_e can infinitely many times reach ($\mathcal{F}(s) \subseteq \{1, \ldots, k\}$ denotes all accepting conditions that s belongs to).

The differences of Algorithm 1 from GS are the following: (i) Algorithm 1 does not stop immediately when an accepting SCC is found (line 18), but continues in the construction, (ii) in lines 25–26 (which correspond to leaving a possible SCC), the states popped from the stack `act` are classified to be either useful (then they are added to Q') or useless (then they are added to `emp`), and (iii) we use Q' and `emp` in lines 10–11 to check whether we already know whether t has a non-empty language. The algorithm returns $\mathcal{A}$ projected to the states Q', which are known to have non-empty languages. Note that Algorithm 1 is amenable to on-the-fly traversal of the automaton $\mathcal{A}$, i.e., $\mathcal{A}$ can be provided implicitly.

Proposition 4.1. *Algorithm 1 is correct.*

5 Efficient Complementation of SDBAs

Algorithm 1 can be used for constructing the useful part of the GBA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}})$, which requires an efficient construction of $\overline{\mathcal{B}}$. In this section, we present such a construction for an SDBA $\mathcal{B}$.

We first explain the NCSB algorithm of Blahoudek *et al.* [12] for complementing SDBAs, which is, to the best of our knowledge, the most efficient complementation algorithm for SDBAs up to date. Later, we identify a source of inefficiency and propose a solution that mitigates it.

5.1 The NCSB-Original Algorithm

The *NCSB-Original* algorithm [12] can be viewed as an extension of the classical algorithm for complementing a nondeterministic *finite* automaton using the power set construction (used to determinize the automaton). The extension assigns every state in a *macro-state* one of the labels $\{N, C, S, C{+}B\}$ depending on the component where the state is present (as defined formally later, B is always a subset of C so the label $C{+}B$ means that the state is both in C and B). The labels characterize the expected status of the runs going through the states. To avoid confusion, we will call a run of the complement automaton $\mathcal{A}_C = (Q_C, \delta_C, Q_{I_C}, F_C)$ a *macro-run*. We usually denote states in Q_C as $\widehat{q} = (N, C, S, B)$, where the components have the following intuitive meaning:

N (nondeterministic): If a run of $\mathcal{A}$ touches a state in N, then it is still in the nondeterministic part Q_1.

C (choice): If a run of $\mathcal{A}$ touches a state in C, then it can never leave the set Q_2, but we are not yet sure whether it is an accepting run. Therefore, every time a run in C leaves an accepting state, we nondeterministically guess whether it was the last time the run has touched an accepting state (in which case we move the run to the set S) or not (in which case it remains in C).

S (safe): If a run arrives into S, it can only remain *safe*, i.e., it will touch no more accepting states in the future. In the case the run *is not* safe, it will be blocked in $\mathcal{A}_C$ as soon as it attempts to touch an accepting state, i.e., if $q \in S$ and $\delta_2(q, a) \in F$, then $\delta_C(\widehat{q}, a) = \emptyset$ (there can still be another safe run in some other guess though).

B (breakpoint): The set B is used for tracking that all runs of $\mathcal{A}$ that arrive into Q_2 will eventually become safe. In particular, once B becomes empty (denoting an accepting state), we copy the runs that are currently in C into B, and remove them from B only when they become safe (i.e., when they have been moved to S).

The construction is formally defined as follows.

Definition 5.1 (cf. [12, Section 3.2])**.** Given an SDBA $\mathcal{A} = (Q_1 \cup Q_2, \delta, Q_I, F)$, where Q_1 and Q_2 are defined as in Section 2, its complement automaton $\mathcal{A}_C = (Q_C, \delta_C, Q_{I_C}, F_C)$ is defined as follows:

- $Q_C = \{(N, C, S, B) \in 2^{Q_1} \times 2^{Q_2} \times 2^{Q_2 \setminus F} \times 2^{Q_2} \mid B \subseteq C\}$.
- $Q_{I_C} = \{(Q_1 \cap Q_I, Q_2 \cap Q_I, \emptyset, Q_2 \cap Q_I)\}$.
- $F_C = \{ (N, C, S, B) \in Q_C \mid B = \emptyset \}$.
- δ_C is the transition function $\delta_C \colon Q_C \times \Sigma \to 2^{Q_C}$ such that $(N', C', S', B') \in \delta_C((N, C, S, B), a)$ iff
  1. $N' = \delta_1(N, a)$,
  2. $C' \cup S' = \delta_t(N, a) \cup \delta_2(C \cup S, a)$,
  3. $C' \cap S' = \emptyset$,
  4. $S' \supseteq \delta_2(S, a)$,
  5. $C' \supseteq \delta_2(C \setminus F, a)$, and
  6. $B' = C'$ if $B = \emptyset$, otherwise $B' = \delta_2(B, a) \cap C'$.

Informally, rules 2–5 enforce that (1) the successors of states in S remain in S', (2) the successors of non-accepting states in C remain in C', (3) all accepting states in $\delta_t(N, a) \cup \delta_2(C \cup S, a)$ stay in C', because S' is a set of non-accepting states, and (4) the rest of the states in $\delta_t(N, a) \cup \delta_2(C \cup S, a)$ are nondeterministically partitioned into C' and S'.

We note that the original definition [12] used yet another condition: "for all $q \in C \setminus F$ it holds that $\delta_2(q, a) \neq \emptyset$." Since we assume the input BA to be complete (cf. Section 2), the condition always holds and hence we drop it. Also note that in order for the result of the NCSB algorithm to be complete, we may need to add a sink state (we hide this from the algorithm to make the presentation clearer). When talking about the size of the set of states or transitions, we only consider those states and transitions reachable from Q_{I_C}.

The best way to get an intuition about the algorithm is to simulate both accepting and rejecting runs of $\mathcal{A}$ in $\mathcal{A}_C$. Let $\rho = q_0 q_1 \ldots q_i \ldots$ be an accepting run of $\mathcal{A}$ over some word $w \in \Sigma^\omega$ and q_i be the first accepting state in ρ. Assume w.l.o.g. that $q_0 \in Q_1$. It is easy to observe that for any macro-run $\Pi = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1) \ldots (N_i, C_i, S_i, B_i) \ldots$, the run ρ is moved from N to C at position i (rule 2 and the fact that S is disjoint with F), i.e., $q_k \in N_k$ for all $1 \leq k \leq i{-}1$ and $q_i \in C_i$. (Moving a run from a set X to another set X' can be achieved by moving the corresponding state from X to X'.) For any $j > i$ with $q_j \in F$, we have the following two cases (nondeterministic guessing by rules 2–5):

- Case (1): The run ρ is moved from C to S at a position $j + 1$. In this case, Π will be blocked later at the position of the next occurrence of an accepting state in ρ (which there are infinitely many), because once ρ has moved to S, it will stay in S (rule 4). It follows that Π is finite and, therefore, not an accepting macro-run.
- Case (2): If we assume that ρ stays in C for all such positions j, then (rule 6) the run ρ will be copied to B the next time B becomes empty (if it ever happens). But then B cannot become empty again because ρ will stay inside it forever (our assumption is that ρ stays in C forever and hence also in B by rule 6). It follows that although Π is infinite, it is not an accepting macro-run.

On the other hand, we can show that if $w \notin \mathcal{L}(\mathcal{A})$, we can construct from its rejecting runs $\rho = q_0 q_1 \ldots q_i \ldots q_j \ldots$ an accepting macro-run $\Pi = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1) \ldots (N_i, C_i, S_i, B_i) \ldots (N_j, C_j, S_j, B_j) \ldots$ of $\mathcal{A}_C$. The strategy of the construction is simple. All such ρ will be moved from N to C at the first occurrence of an accepting state and then to S after the last occurrence of an accepting state. Because all states in $C \cup S \cup B$ can only proceed via deterministic transitions from δ_2, there is only one corresponding run for each of them.

More concretely, we again have two cases: (i) if the run ρ never touches an accepting state, then ρ will stay in N forever (rule 1) and (ii) if q_i and q_j are the first and the last occurrence of an accepting state in ρ (it can happen that $i = j$), then there is a macro-run where ρ is moved from N to C at position i and then moved to S at position $j{+}1$ (rules 2–5). We can show that the following two conditions hold for such a macro-run Π:

1. Π is non-blocking. From the definition of δ_C, a macro-state can become blocking only in one of two cases: (1) the successor of a non-accepting state q in C coincides with the successor of some state in S (rules 3–5) or (2) the successor of a state in S is accepting (by definition of S and rule 4). The case (2) will never happen since Π moves a run to S only after the last occurrence of an accepting state. Suppose that the case (1) happens. Then, it means that there is no accepting state after q in the corresponding run, i.e., q is the successor of the last accepting state, which should already have been moved to S, leading to a contradiction. So Π is non-blocking.

2. Π contains infinitely many accepting macro-states. Starting from any macro-state of Π, no new runs can be moved to B until it becomes empty (rule 6). Since all runs ρ of $\mathcal{A}$ on w are rejecting, they will be moved to S eventually after the last accepting state, i.e., no run can stay in B forever. The set B will eventually become empty and will be reset to C (rule 6). This will occur infinitely often.

5.2 Eager Guessing as the Source of Inefficiency

In this section, we show that complement automata constructed using NCSB-Original are unnecessarily large. Consider the example of an SDBA and its complement in Figure 3 (the figure shows only interesting parts of the automata).

Observe that in Figure 3b, the NCSB-Original algorithm made a guess at the macro-state $\boxed{\emptyset, \{q'_1, q'_2\}, \{q'''_3\}, \emptyset}$. In fact, the construction needs to know whether ρ is in S or C only for the purpose of deciding whether a macro-state is accepting or rejecting (recall that $B \subseteq C$). In Figure 3b, we can find several macro-states (shown as □) that are redundant because the guessing of whether to keep a run ρ in C or move it to S was performed too eagerly.

A good point to do this guessing is to wait for B to become empty; before that we can simply keep the runs in $C \cup S$ in the same set (in Figure 3b, we keep all of them in C in the left-most branch of the complement automaton). If we do so, then none of the □ macro-states needs to be constructed. Note that their successors can be reconstructed from the macro-state $\boxed{\emptyset, \{q'_1, q'_2\}, \{q'''_3\}, \emptyset}$. Having arrived at this macro-state, the guessing of all states in C have been postponed and hence any of them can nondeterministically either stay in C or be moved to S (dashed lines in Figure 3b).

To achieve the effect of delaying the guessing, our first attempt is to redefine the successor relation δ_C from Definition 5.1 such that $(N', C', S', B') \in \delta_C((N, C, S, B), a)$ iff

1. $N' = \delta_1(N, a)$,
2. $C' \cup S' = \delta_t(N, a) \cup \delta_2(C \cup S, a)$,
3. $C' \cap S' = \emptyset$,
4. **[new]** $S' \supseteq \delta_2(S, a)$ if $B = \emptyset$, else $S' = \delta_2(S, a)$, and
5. **[removed]**
6. $B' = C'$ if $B = \emptyset$, otherwise $B' = \delta_2(B, a) \cap C'$

In particular, rule 4 has been exchanged for a new one and rule 5 has been removed. The new rule 4 enforces that all runs that are in S remain there and no new runs are added into S until an accepting macro-state (a macro-state where $B = \emptyset$) is encountered. Additionally, rule 5 from Definition 5.1 has been removed, so now one can nondeterministically move any non-accepting states from C to S when B becomes empty. The justification is that any run ρ in C must have had its guessing postponed (recall that if a run is in C, it must have seen at least one accepting state) and in NCSB-Original could have been moved to S by now. A complement automaton constructed using NCSB-Original will have macro-runs corresponding to every postponed guessing, i.e., macro-runs traversing □ macro-states in Figure 3b. Those macro-runs eventually reach successors of accepting macro-states produced by the modified algorithm, which in Figure 3b correspond to destinations of the dashed transitions.

Unfortunately, the change proposed above is not yet correct due to the issue that some run ρ in B has no chance to be moved to S, even for the case when ρ has no accepting states after the state in B. In such a case, ρ should correspond to an accepting run in the complement automaton $\mathcal{A}_C$, but B can never become empty. This can be fixed by allowing the move of the successors of accepting states in B to S nondeterministically, i.e., guessing that it is the last occurrence of an accepting state in the run. This leads to an algorithm with lazy guessing, which we provide in the following section.

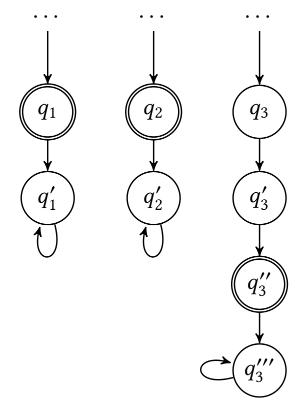
5.3 The NCSB-Lazy Algorithm

Combining the observations in the previous section, we obtain a new SDBA complementation algorithm, called *NCSB-Lazy*. The algorithm is obtained by redefining the transition function δ_C from Definition 5.1 such that $(N', C', S', B') \in \delta_C((N, C, S, B), a)$ iff

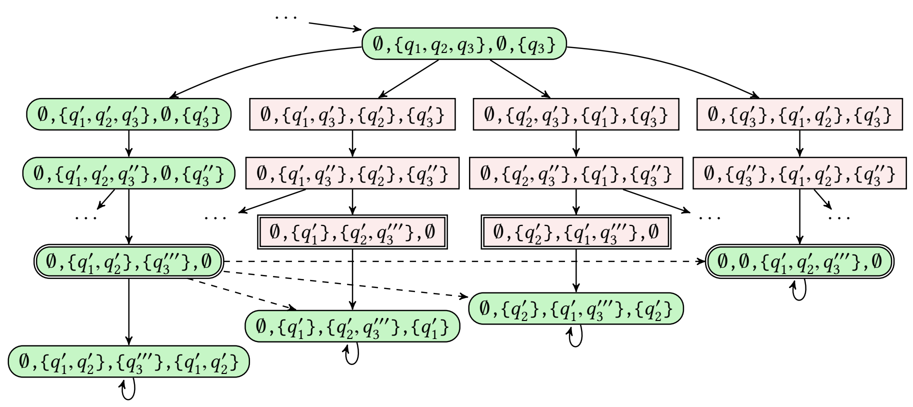
- When $B = \emptyset$:
  - a1. $N' = \delta_1(N, a)$,
  - a2. $C' \cup S' = \delta_t(N, a) \cup \delta_2(C \cup S, a)$,
  - a3. $C' \cap S' = \emptyset$,
  - a4. $S' \supseteq \delta_2(S, a)$, and
  - a5. **[removed]**
  - a6. $B' = C'$.
- When $B \neq \emptyset$:
  - b1. $N' = \delta_1(N, a)$,
  - b2. **[new]** $B' \cup S' = \delta_2(B \cup S, a)$,
  - b3. **[new]** $B' \cap S' = \emptyset$,
  - b4. $S' \supseteq \delta_2(S, a)$,
  - b5. **[new]** $C' = (\delta_2(C, a) \cup \delta_t(N, a)) \setminus S'$, and
  - b6. **[new]** $B' \supseteq \delta_2(B \setminus F, a)$.

When $B = \emptyset$, the construction works in the same way as the one presented in Section 5.2. When $B \neq \emptyset$, rules b2–b4 and b6 enforce that (1) a run in B that touches an accepting state can be nondeterministically moved to S and (2) a run in S will remain in S forever. Rule b5 enforces that if a state is moved from B to S, then it should also be removed from C.

Proposition 5.2. *The complement BA constructed by NCSB-Lazy contains at most as many macro-states as the complement BA constructed by NCSB-Original.*



(a) A part of $\mathcal{A}$ (we assume that all shown states are in Q_2).


$\emptyset, \{q_1, q_2, q_3\}, \emptyset, \{q_3\}$
$\emptyset, \{q_1', q_2', q_3'\}, \emptyset, \{q_3'\}$
$\emptyset, \{q_1', q_3'\}, \{q_2'\}, \{q_3'\}$
$\emptyset, \{q_2', q_3'\}, \{q_1'\}, \{q_3'\}$
$\emptyset, \{q_3'\}, \{q_1', q_2'\}, \{q_3'\}$
$\emptyset, \{q_1', q_2', q_3''\}, \emptyset, \{q_3''\}$
$\emptyset, \{q_1', q_3''\}, \{q_2'\}, \{q_3''\}$
$\emptyset, \{q_2', q_3''\}, \{q_1'\}, \{q_3''\}$
$\emptyset, \{q_3''\}, \{q_1', q_2'\}, \{q_3''\}$
$\emptyset, \{q_1'\}, \{q_2', q_3'''\}, \emptyset$
$\emptyset, \{q_2'\}, \{q_1', q_3'''\}, \emptyset$
$\emptyset, \{q_1', q_2'\}, \{q_3'''\}, \emptyset$
$\emptyset, \emptyset, \{q_1', q_2', q_3'''\}, \emptyset$
$\emptyset, \{q_1'\}, \{q_2', q_3'''\}, \{q_1'\}$
$\emptyset, \{q_2'\}, \{q_1', q_3'''\}, \{q_2'\}$
$\emptyset, \{q_1', q_2'\}, \{q_3'''\}, \{q_1', q_2'\}$

(b) A part of the complemented automaton $\mathcal{A}_C$. Here we only draw macro-states (N, C, S, B) starting from $(\emptyset, \{q_1, q_2, q_3\}, \emptyset, \{q_3\})$ and omit the macro-states that keep q_3''' in C.

Figure 3. An example of inefficiency of eager guessing in NCSB-Original (we assume all transitions are over the symbol $a \in \Sigma$)

Below we give a lemma that will be used in the correctness proof of NCSB-Lazy.

Lemma 5.3. *Consider an SDBA $\mathcal{A}$, its complement $\mathcal{A}_C$ constructed by NCSB-Lazy, a word $w \in \Sigma^\omega$, and a macro-state $\widehat{p} = (N, C, S, B)$ from $\mathcal{A}_C$. Further, assume that for all runs ρ over w in $\mathcal{A}$, it holds that*

1. *if ρ starts from a state $q \in N \cup C \cup S$, it is rejecting,*
2. *if ρ starts from a state $q \in S$, it is safe, and*
3. *if ρ starts from a state $q \in B$, it is not safe.*

Then one can construct an accepting macro-run over w in $\mathcal{A}_C$.

Proof. Our strategy for constructing an accepting macro-run Π from $\widehat{p}$ is the following. If $B = \emptyset$, we move all safe runs in C into S and copy all unsafe runs to the B component of the next macro-state. If $B \neq \emptyset$, we move all runs in B into S immediately when they become safe, i.e., immediately after they touch an accepting state for the last time. The other parts of the construction of Π are deterministic, i.e., one can construct deterministically every macro-state in Π following the transition relation of NCSB-Lazy. We now show that Π is an accepting macro-run in $\mathcal{A}_C$ by proving two properties.

1. Π is non-blocking. From the definition of the transition relation of NCSB-Lazy, a macro-state can become blocking only in one of the following cases: (1) the successor of a non-accepting state q in B coincides with the successor of some state in S (rules b3, b4, and b6) and (2) the successor of a state in S is accepting (by definition of S and rule b4). The case (2) will never happen since, as defined above, Π moves a run to S only after the last occurrence of an accepting state. Suppose that case (1) happens. Then, it means that there is no accepting state after q in the corresponding run, i.e., q is the successor of the last accepting state, which should have already been moved to S, leading to a contradiction.

2. Π contains infinitely many accepting macro-states. Starting from any macro-state of Π, no new runs can be moved to B until it becomes empty (rule a6). Since all runs ρ of $\mathcal{A}$ on w are rejecting starting from any states in $\widehat{p}$, they will be moved to S eventually after the last accepting state, i.e., no run can stay in B forever. The set B will eventually become empty and will (infinitely often) be reset to C (rule a6). □

Theorem 5.4. *Given an SDBA $\mathcal{A}$, NCSB-Lazy produces a BA $\mathcal{A}_C$ such that $\mathcal{L}(\mathcal{A}_C) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

Proof. The case that $w \in \mathcal{L}(\mathcal{A})$ implies $w \notin \mathcal{L}(\mathcal{A}_C)$ can be proved in a similar way as in NCSB-Original. In particular, we need to show that any accepting run ρ of $\mathcal{A}$ over w will either stay forever in B or move to S and block the macro-run.

For the case that $w \notin \mathcal{L}(\mathcal{A})$ implies $w \in \mathcal{L}(\mathcal{A}_C)$, we can construct an accepting macro-run Π from the runs ρ of $\mathcal{A}$ on w using Lemma 5.3. In order to do so, we need to ensure that the initial macro-state $(Q_1 \cap Q_I, Q_2 \cap Q_I, \emptyset, Q_2 \cap Q_I)$ satisfies the requirements of Lemma 5.3, i.e., for all runs ρ over w in $\mathcal{A}$, it holds that

1. if ρ starts from a state $q \in N \cup C \cup S$, it is rejecting,
2. if ρ starts from a state $q \in S$, it is safe, and
3. if ρ starts from a state $q \in B$, it is not safe.

Requirement 1 is satisfied because w has only rejecting runs from the initial states Q_I. Requirement 2 is satisfied because $S = \emptyset$. Requirement 3 is satisfied because all states in $Q_2 \cap Q_I$ are also in F (due to our restriction on SDBAs, cf. Section 2), so runs starting from them are not safe. □

From the experimental results (cf. Section 7), one can see that although the changes in the algorithm are small, they induce a large difference in performance. We believe that the idea of delaying nondeterministic choices can be useful in other algorithms, such as rank-based BA complementation.

6 Subsumption-based Pruning in the Construction of a Difference Automaton

In this section we describe subsumption relations that can be used to optimize the construction of the difference automaton described in Section 4. The subsumption relations are, in a way, similar to the so-called *antichain* [2, 23] algorithms used in language inclusion and universality testing over nondeterministic finite automata.

We start by describing the notation used in this section. For macro-states $\widehat{p} = (N_p, C_p, S_p, B_p)$ and $\widehat{r} = (N_r, C_r, S_r, B_r)$, we define the following two subsumption relations:

$$\widehat{p} \sqsubseteq \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad N_p \supseteq N_r \wedge C_p \supseteq C_r \wedge S_p \supseteq S_r \quad \text{and} \tag{4}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \widehat{p} \sqsubseteq \widehat{r} \wedge B_p \supseteq B_r. \tag{5}$$

Let $\mathcal{A}_C^O$ and $\mathcal{A}_C^L$ be the complement automata constructed using NCSB-Original and NCSB-Lazy, respectively. We define two language inclusion relations $\subseteq_{\mathcal{L}}^O$ and $\subseteq_{\mathcal{L}}^L$ over macro-states $\widehat{p}$ and $\widehat{r}$ as follows:

$$\widehat{p} \subseteq_{\mathcal{L}}^O \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathcal{L}_{\mathcal{A}_C^O}(\widehat{p}) \subseteq \mathcal{L}_{\mathcal{A}_C^O}(\widehat{r}), \tag{6}$$

$$\widehat{p} \subseteq_{\mathcal{L}}^L \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathcal{L}_{\mathcal{A}_C^L}(\widehat{p}) \subseteq \mathcal{L}_{\mathcal{A}_C^L}(\widehat{r}). \tag{7}$$

The main result of this section is that for any macro-states $\widehat{p}$ and $\widehat{r}$, the following implications hold:

$$\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p} \subseteq_{\mathcal{L}}^O \widehat{r} \qquad \text{(Section 6.1) and} \tag{8}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \implies \widehat{p} \subseteq_{\mathcal{L}}^L \widehat{r} \qquad \text{(Section 6.2).} \tag{9}$$

As a consequence, we can use $\sqsubseteq$ and $\sqsubseteq^B$ in Algorithm 1 (when computing the difference automaton $\mathcal{A} \setminus \mathcal{B}$) for early termination when checking whether a language of an encountered macro-state is empty (lines 3 and 11). In particular, we change testing (non-)membership of a macro-state $\widehat{q}$ in the set emp into testing the same in the set $\lceil \text{emp} \rceil$ defined as

$$\lceil \text{emp} \rceil = \{(q_{\mathcal{A}}, \widehat{q_{\mathcal{B}}}) \mid \exists (q_{\mathcal{A}}, \widehat{r_{\mathcal{B}}}) \in \text{emp} : \widehat{q_{\mathcal{B}}} \sqsubseteq' \widehat{r_{\mathcal{B}}}\}, \tag{10}$$

where $\sqsubseteq' \in \{\sqsubseteq, \sqsubseteq^B\}$ depending on the particular algorithm used for complementation ($\sqsubseteq$ for NCSB-Original and $\sqsubseteq^B$ for NCSB-Lazy). Note that on line 26, emp can be maintained in the form of an antichain, i.e., to contain only elements incomparable w.r.t. $\sqsubseteq'$.

6.1 Subsumption Relation for NCSB-Original

We first show that $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p} \subseteq_{\mathcal{L}}^O \widehat{r}$ for any two macro-states $\widehat{p} = (N_p, C_p, S_p, B_p)$ and $\widehat{r} = (N_r, C_r, S_r, B_r)$. We prove this fact by constructing a *strategy* that for any accepting macro-run from $\widehat{p}$ returns an accepting macro-run from $\widehat{r}$ over the same word. Our proof consists of two parts. First, we define two new notions of simulation relation, named *early simulations*, between traces and states of a BA and we show that they under-approximate language inclusion. Second, we prove that both subsumption relations $\sqsubseteq$ and $\sqsubseteq^B$ are instances of the corresponding early simulation relations.

6.1.1 Early Simulation

Consider a BA $\mathcal{A} = (Q, \delta, Q_I, F)$ and a pair of traces $\pi_p = p_0 \xrightarrow{w_0} p_1 \xrightarrow{w_1} \cdots$ and $\pi_r = r_0 \xrightarrow{w_0} r_1 \xrightarrow{w_1} \cdots$ over the word $w = w_0 w_1 \ldots \in \Sigma^\omega$ from the states $p_0 \in Q$ and $r_0 \in Q$. We say that π_p is *early simulated* by π_r (or, alternatively, that π_r early simulates π_p), denoted as $\pi_p \preceq_e \pi_r$, iff

$$\forall i < j : ((p_i \in F \vee i = -1) \wedge p_j \in F) \implies \exists i < k \leq j : r_k \in F, \tag{11}$$

and that π_p is *early+1 simulated* by π_r (written $\pi_p \preceq_{e^{+1}} \pi_r$) iff

$$\forall i < j : (p_i, p_j \in F) \implies \exists i < k \leq j : r_k \in F. \tag{12}$$

Intuitively, the *early+1* simulation requires that between every two times π_p touches an accepting state, π_r also touches an accepting state; the *early* simulation further requires that π_r first touches an accepting state not later than π_p does.

We extend the proposed notions of simulation to states as follows. First, we define a *strategy* as a function $\sigma : Q \times (Q \times \Sigma \times Q) \to (Q \times \Sigma \times Q)$ such that $\sigma(r, p \xrightarrow{a} p') = r \xrightarrow{a} r'$ where $r' \in \delta(r, a)$. That is, σ picks a transition from r based on the transition $p \xrightarrow{a} p'$ selected by the environment. Next, we lift strategy to traces such that for a trace π_p defined as above, we set $\sigma(r_0, \pi_p) = r_0 \xrightarrow{w_0} r_1 \xrightarrow{w_1} \cdots$ where for all $i \geq 0$ it holds that $\sigma(r_i, p_i \xrightarrow{w_i} p_{i+1}) = r_i \xrightarrow{w_i} r_{i+1}$. We say that p_0 is early (resp. early+1) simulated by r_0, denoted as $p_0 \preceq_e r_0$ (resp. $p_0 \preceq_{e^{+1}} r_0$) iff there exists a strategy σ_e (resp. $\sigma_{e^{+1}}$) such that for every trace π_p starting in p_0, it holds that $\pi_p \preceq_e \sigma_e(r_0, \pi_p)$ (resp. $\pi_p \preceq_{e^{+1}} \sigma_{e^{+1}}(r_0, \pi_p)$).

The following proposition states that the introduced simulations under-approximate language inclusion.

Proposition 6.1. *Given a BA $\mathcal{A}$, the following holds for the relations over the states of $\mathcal{A}$:*

$$\preceq_e \quad \subseteq \quad \preceq_{e^{+1}} \quad \subseteq \quad \subseteq_{\mathcal{L}}. \tag{13}$$

6.1.2 The Subsumption $\sqsubseteq$ is an Early Simulation

Consider an SDBA $\mathcal{A}$ and its complement BA $\mathcal{A}_C^O$ constructed by NCSB-Original, and let us fix the following two states of $\mathcal{A}_C^O$: $\widehat{p} = (N_p, C_p, S_p, B_p)$ and $\widehat{r} = (N_r, C_r, S_r, B_r)$.

Lemma 6.2. *The relations $\sqsubseteq$ and $\sqsubseteq^B$ on $\mathcal{A}_C^O$ are an early+1 simulation and an early simulation respectively:*

$$\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p} \preceq_{e^{+1}} \widehat{r} \quad \text{and} \tag{14}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \implies \widehat{p} \preceq_e \widehat{r}. \tag{15}$$

Proof of (14). We use the strategy $\sigma_\sqsubseteq$ that for a transition $\widehat{p} \xrightarrow{a} \widehat{p'} = (N_{p'}, C_{p'}, S_{p'}, B_{p'})$ chooses a transition $\widehat{r} \xrightarrow{a} \widehat{r'} = (N_{r'}, C_{r'}, S_{r'}, B_{r'})$ that respects all nondeterministic choices made in $\widehat{p} \xrightarrow{a} \widehat{p'}$. In particular, if the successor q' of a state $q \in C_p \cap C_r$ was moved to $S_{p'}$, i.e., $q' = \delta_2(q, a) \in S_{p'}$, the strategy $\sigma_\sqsubseteq$ will also move q' to $S_{r'}$, otherwise q' will stay in $C_{r'}$. Other parts of the construction of $\widehat{r'}$ are deterministic (i.e., just follow the definition of δ_C in Definition 5.1). The strategy guarantees that (1) $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p}' \sqsubseteq \widehat{r}'$ and (2) the transition $\widehat{r} \xrightarrow{a} \widehat{r}'$ exists (proof omitted due to lack of space).

Next, we show that for any two traces $\pi_p = \widehat{p_0} \xrightarrow{w_0} \widehat{p_1} \xrightarrow{w_1} \cdots$ and $\pi_r = \sigma_{\sqsubseteq}(r_0, \pi_p) = \widehat{r_0} \xrightarrow{w_0} \widehat{r_1} \xrightarrow{w_1} \cdots$, such that $\widehat{p_0} = \widehat{p}$ and $\widehat{r_0} = \widehat{r}$, the condition $\pi_p \preceq_{e^{+1}} \pi_r$ is satisfied, i.e., $\forall i < j : (\widehat{p_i}, \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O$ holds, where F^O is the set of accepting macro-states of $\mathcal{A}_C^O$.
Claim: For all $i \geq 0$, if $\widehat{p_i} \in F^O$, then $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$.
Proof: Let the macro-states $\widehat{p_{i+1}}$ and $\widehat{r_{i+1}}$ be as follows: $\widehat{p_{i+1}} = (N_{p_{i+1}}, C_{p_{i+1}}, S_{p_{i+1}}, B_{p_{i+1}})$ and $\widehat{r_{i+1}} = (N_{r_{i+1}}, C_{r_{i+1}}, S_{r_{i+1}}, B_{r_{i+1}})$. The following holds: (i) $B_{p_{i+1}} = C_{p_{i+1}}$ (because $\widehat{p_i} \in F^O$), (ii) $C_{p_{i+1}} \supseteq C_{r_{i+1}}$ (due to the property of $\sigma_{\sqsubseteq}$, i.e., $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p}' \sqsubseteq \widehat{r}'$), and (iii) $C_{r_{i+1}} \supseteq B_{r_{i+1}}$ (the property of a macro-state). It follows that $B_{p_{i+1}} \supseteq B_{r_{i+1}}$ and hence $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$. ■
Claim: If $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$ and $\widehat{p_j} \in F^O$ for some $i < j$, then there exists some k such that $i < k \leq j$ and $\widehat{r_k} \in F^O$.
Proof: Since $B_{p_{i+1}} \supseteq B_{r_{i+1}}$ and due to the property of $\sigma_{\sqsubseteq}$ that every state that is moved from $B_{p_{i+1}}$ to $S_{p_{i+1}}$ in π_p will be by $\sigma_{\sqsubseteq}$ also simultaneously moved from $B_{r_{i+1}}$ to $S_{r_{i+1}}$ in π_r, the set $B_{r_{i+1}}$ in π_r will become empty not later than $B_{p_{i+1}}$ becomes empty in π_p. ■

The two claims above imply that $\pi_p \preceq_{e^{+1}} \pi_r$. □

Proof of (15). We use the same strategy $\sigma_{\sqsubseteq}$ from the proof of (14). We show that for any two traces $\pi_p = \widehat{p_0} \xrightarrow{w_0} \widehat{p_1} \xrightarrow{w_1} \cdots$ and $\pi_r = \sigma_{\sqsubseteq}(r_0, \pi_p) = \widehat{r_0} \xrightarrow{w_0} \widehat{r_1} \xrightarrow{w_1} \cdots$, it follows that $\pi_p \preceq_e \pi_r$, i.e., $\forall i < j : ((\widehat{p_i} \in F^O \vee i = -1) \wedge \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O$, where F^O is the set of accepting macro-states of $\mathcal{A}_C^O$. First, we change $\pi_p \preceq_e \pi_r$ into an equivalent conjunction of the following two conditions:

$$\forall i < j : (\widehat{p_i}, \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O, \quad (16)$$

$$\widehat{p_i} \in F^O \implies \exists k \leq i : \widehat{r_k} \in F^O. \quad (17)$$

We notice that Condition (16) is equivalent to $\pi_p \preceq_{e^{+1}} \pi_r$ and since $\sqsubseteq^B$ is stronger than $\sqsubseteq$, from (14) it follows that $\widehat{p} \preceq_{e^{+1}} \widehat{r}$, and because the strategy $\sigma_{\sqsubseteq}$ in the proof of (14) is the same, Condition (16) also holds. Condition (17), on the other hand, follows from the second claim in the proof of (14). □

The following theorem states that $\sqsubseteq$ and $\sqsubseteq^B$ are subsumption relations over the macro-states of $\mathcal{A}_C^O$.

Theorem 6.3. *The relations $\sqsubseteq^B$ and $\sqsubseteq$ under-approximate language inclusion of macro-states in a complement automaton constructed using NCSB-Original:*

$$\sqsubseteq^B \quad \subseteq \quad \sqsubseteq \quad \subseteq \quad \subseteq_{\mathcal{L}}^O. \quad (18)$$

Proof. Follows from Proposition 6.1 and Lemma 6.2. □

6.2 Subsumption relation for NCSB-Lazy

The BAs produced by NCSB-Lazy are different from the BAs produced by NCSB-Original. This, in particular, means that the subsumption relation $\sqsubseteq$ does not under-approximate the language inclusion $\subseteq_{\mathcal{L}}^L$ in BAs produced by NCSB-Lazy.

Remark: Let $\mathcal{A}_C^L$ by a BA produced by NCSB-Lazy from $\mathcal{A}$ and $\widehat{p}$ and $\widehat{r}$ be a pair of macro-states of $\mathcal{A}_C^L$. In general, $\widehat{p} \sqsubseteq \widehat{r}$ does not imply $\widehat{p} \subseteq_{\mathcal{L}}^L \widehat{r}$. In particular, let q be a non-accepting state of $\mathcal{A}$ with only one outgoing transition that is a self-loop and let $\widehat{p} = (\emptyset, \{q\}, \emptyset, \emptyset)$ and $\widehat{r} = (\emptyset, \{q\}, \emptyset, \{q\})$. Note that $\widehat{p} \sqsubseteq \widehat{r}$ (as $\sqsubseteq$ does not relate the B components). Also note that there exists an accepting macro-run $\widehat{p} \cdot (\emptyset, \emptyset, \{q\}, \emptyset)^\omega$ from $\widehat{p}$, while there exists no accepting macro-run from $\widehat{r}$, since the B component of $\widehat{r}$ can never become empty (cf. condition b6 in Section 5.3). □

Fortunately, the stronger subsumption relation $\sqsubseteq^B$ still under-approximates the language inclusion $\subseteq_{\mathcal{L}}^L$, so it can be used to optimize the difference automaton construction.

Theorem 6.4. *The relation $\sqsubseteq^B$ under-approximates language inclusion of macro-states in a complement automaton constructed using NCSB-Lazy:*

$$\sqsubseteq^B \quad \subseteq \quad \subseteq_{\mathcal{L}}^L. \quad (19)$$

Proof. Let $\widehat{p} = (N, C, S, B)$ and $\widehat{r} = (N', C', S', B')$ s.t. $\widehat{p} \sqsubseteq^B \widehat{r}$. For any word $w \in \mathcal{L}_{\mathcal{A}_C^L}(\widehat{p})$, it is not difficult to observe that

- all runs over w from states in $N \cup C \cup S$ are rejecting and the runs from states in $N' \cup C' \cup S'$ are their subset,
- all runs over w from states in S are safe, and the runs from states in S' are their subset, and
- all runs over w from states in B are unsafe, and the runs from states in B' are their subset.

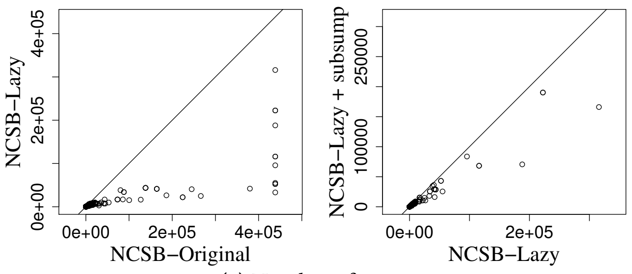
Hence, we can apply Lemma 5.3 to obtain an accepting macro-run from $\widehat{r}$ over w in $\mathcal{A}_C^L$. It follows that $\widehat{p} \subseteq_{\mathcal{L}}^L \widehat{r}$. □

7 Experimental Evaluation

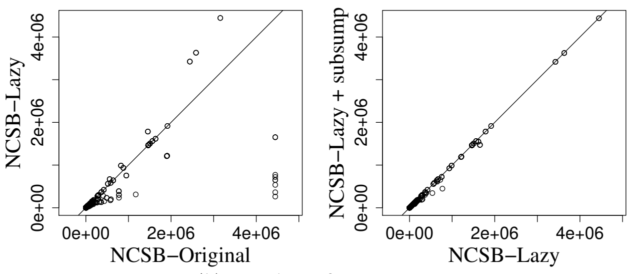
We implemented the presented techniques as an extension of Ultimate Automizer [33] and experimentally evaluated their performance. The results are presented in Figures 4 and 5. The points in the top/right-most regions of the plot represent experiments where the corresponding setting timed out (>300 s) or went out of available memory (4 GiB).

In the first set of experiments, shown in Figure 4, we evaluated the performance of three versions of the SDBA complementation algorithm: NCSB-Original from [12], NCSB-Lazy (Section 5.3), and NCSB-Lazy with subsumption (Section 6.2). We used the set of all 1159 SDBAs produced by Ultimate Automizer during termination analysis of all non-recursive benchmarks (1375 programs) in the Termination category of SV-Comp [1, 10][2].

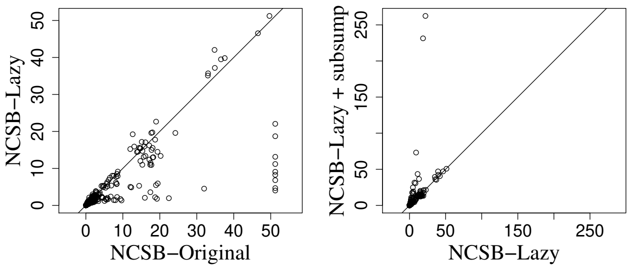
[2]We used 6 different settings and collected all SDBAs produced before the timeout. Because Ultimate Automizer constructs the difference of



(a) Number of states



(b) Number of transitions



(c) Execution time [s]

Figure 4. Comparing the performance of NCSB-Lazy with NCSB-Original and evaluating the effect of subsumption

Figure 4a shows that NCSB-Lazy significantly improves the number of states of the complemented automata, and that the subsumption can save even more states. In Figure 4b, we see that in the majority of cases, NCSB-Lazy also reduces the number of transitions. This is not guaranteed though; in several cases, the number of transitions increased. Subsumption is also not so helpful in reducing the number of transitions. In Figure 4c, we observe that in most cases, NCSB-Lazy also reduces the execution time. On the other hand, subsumption does not help that much as it brings significant overhead. Nevertheless, subsumption always produces fewer states in the BA language difference operation, which is an important factor for the overall performance of the termination analysis. More precisely, the average numbers of **S**tates and **T**ransitions for the three settings are the following:

NCSB-Original:	4,700 S	and	122,200 T
NCSB-Lazy:	2,900 S	and	132,300 T
NCSB-Lazy + Subsumption:	1,600 S	and	111,700 T

automata on the fly, if the construction does not finish before the timeout, the SDBA is not fully built and so cannot be used for this experiment.

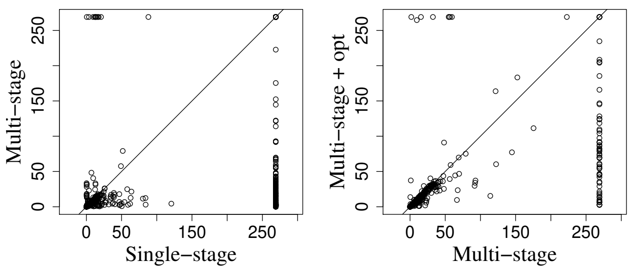


Figure 5. Evaluating the performance of the multi-stage approach and the optimized difference operation [s]

In the next experiment, we evaluated the performance of the proposed optimizations within program termination analysis. We again use all non-recursive programs from the Termination category of SV-Comp. In the left-hand side of Figure 5, we evaluated the performance of the multi-stage approach w.r.t. the single-stage approach (which always directly generalizes a counterexample to a nondeterministic module $\mathcal{M}_{nondet}$). For the multi-stage approach, we first evaluated three different generalization sequences:

(i) $\mathcal{M}_{uv^\omega} \to \mathcal{M}_{\text{fin}} \to \mathcal{M}_{\text{semi}} \to \mathcal{M}_{nondet}$ (we skip $\mathcal{M}_{\text{det}}$)
(ii) $\mathcal{M}_{uv^\omega} \to \mathcal{M}_{\text{fin}} \to \mathcal{M}_{\text{det}} \to \mathcal{M}_{nondet}$ (we skip $\mathcal{M}_{\text{semi}}$)
(iii) $\mathcal{M}_{uv^\omega} \to \mathcal{M}_{\text{fin}} \to \mathcal{M}_{\text{det}} \to \mathcal{M}_{\text{semi}} \to \mathcal{M}_{nondet}$

All of them solved roughly the same amount of examples (±2 in the set of 1375 programs) when the SDBA difference optimization was not used. Therefore, we chose option (i), which produces the most SDBAs, so we can exploit the full potential of our optimizations. Using this option, the analysis of 1375 programs generated 6375 finite-trace modules, 1200 semideterministic modules, and 3 nondeterministic modules. We can see that the multi-stage approach solves significantly more cases than the single-stage approach (fewer points in the up-most region of the plot). The improvement is obtained mainly by avoiding the construction of $\mathcal{M}_{nondet}$, which has a costly complementation procedure. The occasional slowdown can still happen since different counterexample generalization constructions produce BAs with different languages (in the subsequent steps, we then obtain different counterexamples, giving rise to a different global search space).

In the right-hand side of Figure 5, we evaluated the performance of the proposed optimizations of the difference automaton construction ("Multi-stage + opt" uses NCSB-Lazy + subsumption to complement SDBAs). We can observe that there are some cases where the version with optimizations has a worse time or even times out, but the version without optimizations can solve them. This can happen because of one of the following reasons: (1) the subsumption techniques impose overhead on the execution time or (2) NCSB-Lazy produces more transitions than NCSB-Original. We can, however, clearly see that the proposed optimizations are indeed helpful in the overall performance of the termination analysis, as the number of solved cases is significantly higher than for the version without them.

In particular, the number of benchmarks that timeouted or ran out of memory is for the various settings as follows:

Single-stage:	691
Multi-stage without optimizations:	296
Multi-stage with Subsumption:	253
Multi-stage with NCSB-Lazy:	250
Multi-stage with NCSB-Lazy + Subsumption:	249

One can see that both NCSB-Lazy and Subsumption are already quite useful to improve the overall performance, but the best result is obtained by turning all optimizations on.

8 Related Work

To the best of our knowledge, there is no other tool implementing a termination analysis closely related to the algorithm implemented in Ultimate Automizer. Hence, our evaluation in Section 7 was focussed on different variations of this algorithm. For a comparison with other tools, we rather refer to the results of the independent competition on software verification SV-Comp. In SV-Comp 2018 [10] the Ultimate Automizer team used the optimizations that were presented in this paper and won the Termination category.[3] For the sake of completeness, we give a brief overview of other termination analyses that make use of automata or that have participated in SV-Comp.

One line of research is based on the *size-change principle* [4, 5, 37]. In this technique, one examines the flow of values among variables in each code block, which values are bounded from below (e.g., by the condition of an `if` statement), which values are not increasing, and which values are decreasing. The basic idea of this approach is that if on each (infinite) path there is at least one value that is bounded from below, never increasing, and infinitely often decreasing, then the program terminates. This property is inferred using one of two techniques. One is based on BAs, the other is based on Ramsey's theorem. In contrast to our approach, there is only one BA, in general not semideterministic. The BA is, however, reverse-deterministic, which also allows a more efficient language inclusion check [24]. Although this approach and the approach implemented in Ultimate Automizer both use BAs, they are not closely connected. Using size-change termination, one always has a fixed domain of constraints, which is used to track decreasing values, whereas in Ultimate Automizer we may use ranking functions to track the values of arbitrary expressions. Furthermore, Ultimate Automizer infers the ranking functions on-demand and splits the program into components where these ranking functions serve as termination proofs.

Another line of research is based on *transition invariants* [45]. There, the basic idea is that if the transitive closure of the program's transition relation is a subset of a union of well-founded relations, then the program is terminating. In this line of research, soundness is also proved via Ramsey's theorem. The approach has been first implemented in the Terminator tool [17], and later also in T2 [15] and CPAchecker [11, 41]. The set of well-founded relations used there is obtained from a set of functions, where each of them is a ranking function for a set of paths in the program. Terminator constructs this set of functions incrementally in the following CEGAR-style algorithm. The tool lets a safety checker analyze if, in each loop location, at least one of the functions is a ranking function. If not, a lasso-shaped counterexample is obtained and its termination is analyzed by techniques specialized for lasso-shaped programs [6–9, 14, 16, 32, 39, 44]. If the counterexample is spurious, i.e., the lasso-shaped program is terminating, a ranking function of this paths is constructed and added to the set of functions. This process is repeated until a real counterexample is found or the safety checker detected that, for each path, one of the functions is a ranking function. A bottleneck of this approach is that the safety checks become costlier over time since the set of ranking functions is growing. In Ultimate Automizer, this bottleneck is shifted from a program analysis task to an automata theory task. We never have to combine several ranking functions since the program is decomposed into several modules and there is only one function for each module. This comes at the price that the number of modules is growing over time, so the automata operations that are applied to these modules also become costlier.

The AProVE tool [28] first applies several transformations (e.g., removing pointers [48]) to translate a program into an integer term rewriting system. Afterwards, it applies various techniques to analyze termination of the resulting system [25, 29]. Termination can also be analyzed via an abstract interpretation framework [21]. Several abstract domains have been developed [20, 49, 52, 53] and implemented in the FuncTion tool [50]. In contrast to Ultimate Automizer, which is decomposing the set of program traces, there are also tools that decompose the state space of the program, such as HipTNT+ [36] and SeaHorn [51]. Decomposing the state space allows them to infer ranking functions for each component separately.

Acknowledgments

We thank the reviewers for their suggestions, which significantly helped to improve the readability of the paper. The work on this paper was supported by the Czech Science Foundation project 17-12465S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), the FIT BUT project FIT-S-17-4014, the National Natural Science Foundation of China (Grants No. 61532019, 61650410658, 61761136011), the CAS/SAFEA International Partnership Program for Creative Research Teams, the CDZ project CAP (GZ 1023), and the Ministry of Science and Technology of Taiwan (project 106-2221-E-001-009-MY3).

[3]https://sv-comp.sosy-lab.org/2018/results/results-verified/

References

[1] Software Verification Competition (SV-Comp) Benchmarks. https://github.com/sosy-lab/sv-benchmarks. Accessed: 2017-11-01.

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. 2010. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In *Proceedings of 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. Springer-Verlag, Berlin, Heidelberg, 158–174. https://doi.org/10.1007/978-3-642-12002-2_14

[3] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.

[4] Amir M. Ben-Amram. 2010. Size-Change Termination, Monotonicity Constraints and Ranking Functions. *Logical Methods in Computer Science* 6, 3 (2010). http://arxiv.org/abs/1005.0253

[5] Amir M. Ben-Amram. 2011. Monotonicity Constraints for Termination in the Integer Domain. *Logical Methods in Computer Science* 7, 3 (2011). https://doi.org/10.2168/LMCS-7(3:4)2011

[6] Amir M. Ben-Amram and Samir Genaim. 2013. On the Linear Ranking Problem for Integer Linear-Constraint Loops. In *Proceedings of 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2429069.2429078

[7] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55. https://doi.org/10.1145/2629488

[8] Amir M. Ben-Amram and Samir Genaim. 2015. Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions. In *Proceedings of 27th International Conference on Computer Aided Verification (CAV'15)*, Vol. 9207. Springer, Cham, 304–321. https://doi.org/10.1007/978-3-319-21668-3_18

[9] Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *Proceedings of 29th International Conference on Computer Aided Verification (CAV'17)*, Vol. 10427. Springer, Cham, 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

[10] Dirk Beyer. 2017. Software Verification with Validation of Results - (Report on SV-Comp 2017). In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Springer-Verlag New York, Inc., New York, NY, USA, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 184–190. http://dl.acm.org/citation.cfm?id=2032305.2032321

[12] František Blahoudek, Matthias Heizmann, Sven Schewe, Jan Strejček, and Ming-Hsien Tsai. 2016. Complementing Semi-deterministic Büchi Automata. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 770–787. https://doi.org/10.1007/978-3-662-49674-9_49

[13] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, Vol. 10205. Springer, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

[14] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 491–504. https://doi.org/10.1007/11513988_48

[15] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, Vol. 9636. Springer, Berlin, Heidelberg, 387–393. https://doi.org/10.1007/978-3-662-49674-9_22

[16] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2013. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design* 43, 1 (2013), 93–120. https://doi.org/10.1007/s10703-013-0186-4

[17] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 415–426. https://doi.org/10.1145/1133981.1134029

[18] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving Program Termination. *Commun. ACM* 54, 5 (2011), 88–98. https://doi.org/10.1145/1941487.1941509

[19] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Proceedings of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

[20] Nathanaël Courant and Caterina Urban. 2017. Precise Widening Operators for Proving Termination by Abstract Interpretation. In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, Vol. 10205. Springer, Berlin, Heidelberg, 136–152. https://doi.org/10.1007/978-3-662-54577-5_8

[21] Patrick Cousot and Radhia Cousot. 2012. An Abstract Interpretation Framework for Termination. In *Proceedings of 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/2103656.2103687

[22] Jean-Michel Couvreur. 1999. On-the-fly Verification of Linear Temporal Logic. In *Proceedings of International Symposium on Formal Methods (FM'99)*. Springer-Verlag, London, UK, UK, 253–271. https://doi.org/10.1007/3-540-48119-2_16

[23] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. 2006. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proceedings of 18th International Conference on Computer Aided Verification (CAV'06)*, Vol. 4144. Springer, Berlin, Heidelberg, 17–30. https://doi.org/10.1007/11817963_5

[24] Seth Fogarty and Moshe Y. Vardi. 2012. Büchi Complementation and Size-Change Termination. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:13)2012

[25] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. 2009. Proving Termination of Integer Term Rewriting. In *Proceedings of 20th International Conference on Rewriting Techniques and Applications (RTA'09)*. Springer-Verlag, Berlin, Heidelberg, 32–47. https://doi.org/10.1007/978-3-642-02348-4_3

[26] Andreas Gaiser and Stefan Schwoon. 2009. Comparison of Algorithms for Checking Emptiness of Büchi Automata. In *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Vol. 13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/2349

[27] Pierre Ganty and Samir Genaim. 2013. Proving Termination Starting from the End. In *Proceedings of 25th International Conference on Computer Aided Verification (CAV'13)*, Vol. 8044. Springer-Verlag New York, Inc., New York, NY, USA, 397–412. https://doi.org/10.1007/978-3-642-39799-8_27

[28] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reasoning* 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[29] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2006. Mechanizing and Improving Dependency Pairs. *J. Autom. Reasoning* 37, 3 (2006), 155–203. https://doi.org/10.1007/s10817-006-9057-7

[30] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Springer. https://doi.org/10.1007/3-540-60761-7

[31] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Proceedings of 17th International Conference on Static Analysis (SAS'10).* Springer-Verlag, Berlin, Heidelberg, 304–319. http://dl.acm.org/citation.cfm?id=1882094.1882113

[32] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. 2013. Linear Ranking for Linear Lasso Programs. In *Proceedings of 15th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, Vol. 8172. Springer, Cham, 365–380. https://doi.org/10.1007/978-3-319-02444-8_26

[33] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Proceedings of 26th International Conference on Computer Aided Verification (CAV'14).* Springer-Verlag New York, Inc., New York, NY, USA, 797–813. https://doi.org/10.1007/978-3-319-08867-9_53

[34] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Proceedings of 22nd International Conference on Computer Aided Verification (CAV'10).* Springer-Verlag, Berlin, Heidelberg, 89–103. https://doi.org/10.1007/978-3-642-14295-6_9

[35] Robert P. Kurshan. 1987. Complementing Deterministic Büchi Automata in Polynomial Time. *J. Comput. Syst. Sci.* 35, 1 (1987), 59–71. https://doi.org/10.1016/0022-0000(87)90036-5

[36] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-termination Specification Inference. In *Proceedings of 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 489–498. https://doi.org/10.1145/2737924.2737993

[37] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01).* ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

[38] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. 2012. Termination Analysis with Algorithmic Learning. In *Proceedings of 24th International Conference on Computer Aided Verification (CAV'12).* Springer-Verlag, Berlin, Heidelberg, 88–104. https://doi.org/10.1007/978-3-642-31424-7_12

[39] Jan Leike and Matthias Heizmann. 2015. Ranking Templates for Linear Loops. *Logical Methods in Computer Science* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:16)2015

[40] Max Michel. 1988. *Complementation is more difficult with automata on infinite words.* Technical Report. CNET, Paris.

[41] Sebastian Ott. 2016. Implementing a Termination Analysis using Configurable Software Analysis. Master's Thesis, University of Passau, Software Systems Lab.

[42] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing Liveness to Safety in First-Order Logic. *ACM Program. Lang.* 2, POPL (2018), 26:1–26:33. https://doi.org/10.1145/3158114

[43] Doron Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *Proceedings of 5th International Conference on Computer Aided Verification (CAV'93).* Springer-Verlag, London, UK, 409–423. http://dl.acm.org/citation.cfm?id=647762.735490

[44] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proceedings of 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, Vol. 2937. Springer, Berlin, Heidelberg, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[45] Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *Proceedings of 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04).* IEEE Computer Society, Washington, DC, USA, 32–41. https://doi.org/10.1109/LICS.2004.50

[46] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. 2008. Heap Assumptions on Demand. In *Proceedings of 20th International Conference on Computer Aided Verification (CAV'08).* Springer-Verlag, Berlin, Heidelberg, 314–327. https://doi.org/10.1007/978-3-540-70545-1_31

[47] Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-threaded Programs. In *Proceedings of 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12).* Springer-Verlag, Berlin, Heidelberg, 237–251. https://doi.org/10.1007/978-3-642-28756-5_17

[48] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. 2017. Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic. *J. Autom. Reasoning* 58, 1 (2017), 33–65. https://doi.org/10.1007/s10817-016-9389-x

[49] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *Proceedings of 24th International Symposium on Static Analysis (SAS'13)*, Vol. 7935. Springer, Berlin, Heidelberg, 43–62. https://doi.org/10.1007/978-3-642-38856-9_5

[50] Caterina Urban. 2015. FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution). In *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15).* Springer-Verlag New York, Inc., New York, NY, USA, 464–466. https://doi.org/10.1007/978-3-662-46681-0_46

[51] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16).* Springer-Verlag New York, Inc., New York, NY, USA, 54–70. https://doi.org/10.1007/978-3-662-49674-9_4

[52] Caterina Urban and Antoine Miné. 2014. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *Proceedings of 23rd European Symposium on Programming Languages and Systems (ESOP'14).* Springer-Verlag New York, Inc., New York, NY, USA, 412–431. https://doi.org/10.1007/978-3-642-54833-8_22

[53] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Proceedings of 21st International Symposium on Static Analysis (SAS'14)*, Vol. 8723. Springer, Cham, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

[54] Antti Valmari. 1991. Stubborn Sets for Reduced State Space Generation. In *Proceedings of 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets (ICATPN'89).* Springer, 491–515. https://doi.org/10.1007/3-540-53863-1_36

International Journal on Software Tools for Technology Transfer (2020) 22:523–539
https://doi.org/10.1007/s10009-019-00520-8

REGULAR PAPER

Approximate reduction of finite automata for high-speed network intrusion detection

Milan Češka[1] · Vojtěch Havlena[1] · Lukáš Holík[1] · Ondřej Lengál[1] · Tomáš Vojnar[1]

Published online: 24 May 2019

Abstract
We consider the problem of *approximate reduction of non-deterministic automata* that appear in hardware-accelerated network intrusion detection systems (NIDSes). We define an error *distance* of a reduced automaton from the original one as the probability of packets being incorrectly classified by the reduced automaton (wrt the probabilistic distribution of packets in the network traffic). We use this notion to design an *approximate reduction procedure* that achieves a great size reduction (much beyond the state-of-the-art language-preserving techniques) with a controlled and small error. We have implemented our approach and evaluated it on use cases from SNORT, a popular NIDS. Our results provide experimental evidence that the method can be highly efficient in practice, allowing NIDSes to follow the rapid growth in the speed of networks.

Keywords Reduction · Nondeterministic finite automata · Deep packet inspection · High-speed network monitoring

1 Introduction

The recent years have seen a boom in the number of security incidents in computer networks. In order to alleviate the impact of network attacks and intrusions, Internet service providers want to detect malicious traffic at their network's entry points and on the backbones between sub-networks. Software-based network intrusion detection systems (NIDSes), such as the popular open-source system SNORT [50], are capable of detecting suspicious network traffic by testing (among others) whether a packet payload matches a regular expression (regex) describing known patterns of malicious traffic. NIDSes collect and maintain vast databases of such regexes that are typically divided into groups according to types of the attacks and target protocols.

Regex matching is the most computationally demanding task of a NIDS as its cost grows with the speed of the network traffic as well as with the number and complexity of the regexes being matched. The current software-based NIDSes cannot perform the regex matching on networks beyond 1 Gbps [5,28], so they cannot handle the current speed of backbone networks ranging between tens and hundreds of Gbps. A promising approach to speed up NIDSes is to (partially) offload regex matching into hardware [27,28,36]. The hardware then serves as a pre-filter of the network traffic, discarding the majority of the packets from further processing. Such pre-filtering can easily reduce the traffic the NIDS needs to handle by two or three orders of magnitude [28].

Field-programmable gate arrays (FPGAs) are the leading technology in high-throughput regex matching. Due to their inherent parallelism, FPGAs provide an efficient way of implementing *non-deterministic finite automata* (NFAs), which naturally arise from the input regexes. Although the amount of available resources in FPGAs is continually increasing, the speed of networks grows even faster. Working with multi-gigabit networks requires the hardware to use many parallel packet processing branches in a single

Ondřej Lengál
lengal@fit.vutbr.cz
http://www.fit.vutbr.cz/∼lengal

Milan Češka
ceskam@fit.vutbr.cz
http://www.fit.vutbr.cz/∼ceskam

Vojtěch Havlena
ihavlena@fit.vutbr.cz
http://www.fit.vutbr.cz/∼ihavlena

Lukáš Holík
holik@fit.vutbr.cz
http://www.fit.vutbr.cz/∼holik

Tomáš Vojnar
vojnar@fit.vutbr.cz
http://www.fit.vutbr.cz/∼vojnar

[1] IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Brno, Czech Republic

FPGA [36]; each of them implementing a separate copy of the concerned NFA, and so reducing the size of the NFAs is of the utmost importance. Various language-preserving automata reduction approaches exist, mainly based on computing (bi)simulation relations on automata states (cf. the related work). The reductions they offer, however, do not satisfy the needs of high-speed hardware-accelerated NIDSes.

Our answer to the problem is *approximate reduction* of NFAs, allowing for a trade-off between the achieved reduction and the precision of the regex matching. To formalize the intuitive notion of precision, we propose a novel *probabilistic distance* of automata. It captures the probability that a packet of the input network traffic is incorrectly accepted or rejected by the approximated NFA. The distance assumes a *probabilistic model* of the network traffic. (We show later how such a model can be obtained.)

Having formalized the notion of precision, we specify the target of our reductions as two variants of an optimization problem: (1) minimizing the NFA size given the maximum allowed error (distance from the original), or (2) minimizing the error given the maximum allowed NFA size. Finding such optimal approximations is, however, computationally hard (**PSPACE**-complete, the same as precise NFA minimization).

Consequently, we sacrifice the optimality and, motivated by the typical structure of NFAs that emerge from a set of regexes used by NIDSes (a union of many long "tentacles" with occasional small strongly connected components), we limit the space of possible reductions by restricting the set of operations they can apply to the original automaton. Namely, we consider two reduction operations: (i) collapsing the future of a state into a *self-loop* (this reduction over-approximates the language), or (ii) *removing states* (such a reduction is under-approximating).

The problem of identifying the optimal sets of states on which these operations should be applied is still **PSPACE**-complete. The restricted problem is, however, more amenable to an approximation by a *greedy algorithm*. The algorithm applies the reductions state-by-state in an order determined by a pre-computed *error labelling* of the states. The process is stopped once the given optimization goal in terms of the size or error is reached. The labelling is based on the probability of packets that may be accepted through a given state and hence over-approximates the error that may be caused by applying the reduction at a given state. As our experiments show, this approach can give us high-quality reductions while ensuring formal error bounds.

Finally, it turns out that even the pre-computation of the error labelling of the states is costly (again **PSPACE**-complete). Therefore, we propose several ways to cheaply over-approximate it such that the strong error bound guarantees are still preserved. In particular, we are able to exploit the typical structure of the "union of tentacles" of the hardware NFA in an algorithm that is exponential in the size of the largest "tentacle" only, which gives us a method that is indeed much faster in practice.

We have implemented our approach and evaluated it on regexes used to classify malicious traffic in Snort. We obtain quite encouraging experimental results demonstrating that our approach provides a much better reduction than language-preserving techniques with an almost negligible error. In particular, our experiments, going down to the level of an actual implementation of NFAs in FPGAs, confirm that we can squeeze into an up-to-date FPGA chip real-life regexes encoding malicious traffic, allowing them to be used with a negligible error for filtering at speeds of 100 Gbps (and even 400 Gbps). This is far beyond what one can achieve with current exact reduction approaches.

This paper is an extended version of the paper that appeared in the proceedings of TACAS'18 [12], containing complete proofs of the presented lemmas and theorems.

Related Work Hardware acceleration for regex matching at the line rate is an intensively studied technology that uses general-purpose hardware [3,4,29–33,49,53] as well as FPGAs [8,14,25,27,28,36,39,45,47]. Most of the works focus on DFA implementation and optimization techniques. NFAs can be exponentially smaller than DFAs but need, in the worst case, $\mathcal{O}(n)$ memory accesses to process each byte of the payload where n is the number of states. In most cases, this incurs an unacceptable slowdown. Several works alleviate this disadvantage of NFAs by exploiting reconfigurability and fine-grained parallelism of FPGAs, allowing one to process one character per clock cycle (e.g. [8,27,28,36,39,45,47]).

In [33], which is probably the closest work to ours, the authors consider a set of regexes describing network attacks. They replace a potentially prohibitively large DFA by a tree of smaller DFAs, an alternative to using NFAs that minimizes the latency occurring in a non-FPGA-based implementation. The language of every DFA-node in the tree over-approximates the languages of its children. Packets are filtered through the tree from the root downwards until they belong to the language of the encountered nodes, and may be finally accepted at the leaves, or are rejected otherwise. The over-approximating DFAs are constructed using a similar notion of probability of an occurrence of a state as in our approach. The main differences from our work are that (1) the approach targets approximation of DFAs (not NFAs), (2) the over-approximation is based on a given traffic sample only (it cannot benefit from a probabilistic model), and (3) no probabilistic guarantees on the approximation error are provided.

Approximation of DFAs was considered in various other contexts. Hyper-minimization is an approach that is allowed to alter language membership of a finite set of words [21,35]. A DFA with a given maximum number of states is constructed in [20], minimizing the error defined either by (i) counting prefixes of misjudged words up to some length, or (ii) the sum of the probabilities of the misjudged words wrt the Poisson distribution over Σ^*. Neither of these approaches considers reduction of NFAs nor allows to control the expected error with respect to the real traffic.

In addition to the metrics mentioned above when discussing the works [20,21,35], the following metrics should also be mentioned. The Cesaro–Jaccard distance studied in [44] is, in spirit, similar to [20] and does also not reflect the probability of individual words. The edit distance of weighted automata from [41] depends on the minimum edit distance between pairs of words from the two compared languages, again regardless of their statistical significance. One might also consider using the error metric on a pair of automata introduced by Angluin in the setting of PAC (probably approximately correct) learning of DFAs [1], where n words are sampled from a given distribution and their (non-)acceptance tested in the two automata. If the outputs of both automata agree on all n words, one can say that with confidence δ the distance between the two automata is at most ϵ, where δ and ϵ can be determined from n. None of these notions is suitable for our needs.

Language-preserving minimization of a given NFA is a **PSPACE**-complete problem [26,34]. More feasible (polynomial time) is language-preserving size reduction of NFAs based on (bi)simulations [9,13,24,42], which does not aim for a truly minimal NFA. A number of advanced variants exist, based on multi-pebble or look-ahead simulations, or on combinations of forward and backward simulations [15,18,37]. The practical efficiency of these techniques is, however, often insufficient to allow them to handle the large NFAs that occur in practice and/or they do not manage to reduce the NFAs enough. Finally, even a minimal NFA for the given set of regexes is often too big to be implemented in the given FPGA operating on the required speed (as shown even in our experiments). Our approach is capable of a much better reduction for the price of a small change of the accepted language.

2 Preliminaries

We use $\langle a, b\rangle$ to denote the set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ and $\mathbb{N}$ to denote the set $\{0, 1, 2, \ldots\}$. Given a pair of sets X_1 and X_2, we use $X_1 \triangle X_2$ to denote their *symmetric difference*, i.e. the set $\{x \mid \exists! i \in \{1, 2\} : x \in X_i\}$. We use the notation $[v_1, \ldots, v_n]$ to denote a vector of n elements, $\mathbf{1}$ to denote the all 1's vector $[1, \ldots, 1]$ (the dimension of $\mathbf{1}$ is always clear from the context), $\boldsymbol{A}$ to denote a matrix, and $\boldsymbol{A}^\top$ for its transpose, and $\boldsymbol{I}$ for the identity matrix.

In the following, we fix a finite non-empty alphabet Σ. A *non-deterministic finite automaton* (NFA) is a quadruple $\mathcal{A} = (Q, \delta, I, F)$ where Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. We use $Q[\mathcal{A}]$, $\delta[\mathcal{A}]$, $I[\mathcal{A}]$, and $F[\mathcal{A}]$ to denote Q, δ, I, and F, respectively, and $q \xrightarrow{a} q'$ to denote that $q' \in \delta(q, a)$. Often, we abuse notation and treat δ as a subset of $Q \times \Sigma \times 2^Q$. A sequence of states $\rho = q_0 \cdots q_n$ is a *run* of A over a word $w = a_1 \cdots a_n \in \Sigma^*$ from a state q to a state q', denoted as $q \overset{w,\rho}{\rightsquigarrow} q'$, if $\forall 1 \leq i \leq n : q_{i-1} \xrightarrow{a_i} q_i$, $q_0 = q$, and $q_n = q'$. Sometimes, we use ρ in set operations where it behaves as the set of states it contains. We also use $q \overset{w}{\rightsquigarrow} q'$ to denote that $\exists \rho \in Q^* : q \overset{w,\rho}{\rightsquigarrow} q'$ and $q \rightsquigarrow q'$ to denote that $\exists w : q \overset{w}{\rightsquigarrow} q'$. The *language* of a state q is defined as $L_{\mathcal{A}}(q) = \{w \mid \exists q_F \in F : q \overset{w}{\rightsquigarrow} q_F\}$ and its *banguage* (back-language) is defined as $L^\flat_{\mathcal{A}}(q) = \{w \mid \exists q_I \in I : q_I \overset{w}{\rightsquigarrow} q\}$. Both notions can be naturally extended to a set $S \subseteq Q$: $L_{\mathcal{A}}(S) = \bigcup_{q \in S} L_{\mathcal{A}}(q)$ and $L^\flat_{\mathcal{A}}(S) = \bigcup_{q \in S} L^\flat_{\mathcal{A}}(q)$. We drop the subscript A when the context is obvious. $\mathcal{A}$ *accepts* the language $L(\mathcal{A})$ defined as $L(\mathcal{A}) = L_{\mathcal{A}}(I)$. $\mathcal{A}$ is called *deterministic* (DFA) if $|I| = 1$ and $\forall q \in Q$ and $\forall a \in \Sigma$: $|\delta(q, a)| \leq 1$, and *unambiguous* (UFA) if $\forall w \in L(\mathcal{A})$: $\exists! q_I \in I, \rho \in Q^*, q_F \in F : q_I \overset{w,\rho}{\rightsquigarrow} q_F$.

The *restriction* of $\mathcal{A}$ to $S \subseteq Q$ is an NFA $\mathcal{A}_{|S}$ given as $\mathcal{A}_{|S} = (S, \delta \cap (S \times \Sigma \times 2^S), I \cap S, F \cap S)$. We define the *trim* operation as $trim(\mathcal{A}) = \mathcal{A}_{|C}$ where $C = \{q \mid \exists q_I \in I, q_F \in F : q_I \rightsquigarrow q \rightsquigarrow q_F\}$. For a set of states $R \subseteq Q$, we use $reach(R)$ to denote the set of states reachable from R, $reach(R) = \{r' \mid \exists r \in R : r \rightsquigarrow r'\}$. We use the number of states of $\mathcal{A}$ as a measure of its size, i.e. $|\mathcal{A}| = |Q|$.

A (discrete probability) *distribution* over a countable set X is a mapping $\Pr : X \to \langle 0, 1\rangle$ such that $\sum_{x \in X} \Pr(x) = 1$. An n-state *probabilistic automaton* (PA) over Σ is a triple $\mathcal{P} = (\boldsymbol{\alpha}, \boldsymbol{\gamma}, \{\boldsymbol{\Delta}_a\}_{a \in \Sigma})$ where $\boldsymbol{\alpha} \in \langle 0, 1\rangle^n$ is a vector of *initial weights*, $\boldsymbol{\gamma} \in \langle 0, 1\rangle^n$ is a vector of *final weights*, and for every $a \in \Sigma$, $\boldsymbol{\Delta}_a \in \langle 0, 1\rangle^{n \times n}$ is a *transition matrix* for symbol a. We abuse notation and use $Q[\mathcal{P}]$ to denote the set of states $Q[\mathcal{P}] = \{1, \ldots, n\}$. Moreover, the following two properties need to hold: (i) $\sum\{\boldsymbol{\alpha}[i] \mid i \in Q[\mathcal{P}]\} = 1$ (the initial probability is 1) and (ii) for every state $i \in Q[\mathcal{P}]$ it holds that $\sum\{\boldsymbol{\Delta}_a[i, j] \mid j \in Q[\mathcal{P}], a \in \Sigma\} + \boldsymbol{\gamma}[i] = 1$. (The probability of accepting or leaving a state is 1.) We define the *support* of $\mathcal{P}$ as the NFA $supp(\mathcal{P}) = (Q[\mathcal{P}], \delta[\mathcal{P}], I[\mathcal{P}], F[\mathcal{P}])$ s.t.

$$\delta[\mathcal{P}] = \{(i, a, j) \mid \boldsymbol{\Delta}_a[i, j] > 0\},$$
$$I[\mathcal{P}] = \{i \mid \boldsymbol{\alpha}[i] > 0\},$$
$$F[\mathcal{P}] = \{i \mid \boldsymbol{\gamma}[i] > 0\}.$$

Let us assume that every PA $\mathcal{P}$ is such that $supp(\mathcal{P}) = trim(supp(\mathcal{P}))$. For a word $w = a_1 \ldots a_k \in \Sigma^*$, we use $\boldsymbol{\Delta}_w$ to denote the matrix $\boldsymbol{\Delta}_{a_1} \cdots \boldsymbol{\Delta}_{a_k}$. For the empty word ε, we define $\boldsymbol{\Delta}_\varepsilon = \boldsymbol{I}$. It can be easily shown that $\mathcal{P}$ represents a distribution over words $w \in \Sigma^*$ defined as $\Pr_{\mathcal{P}}(w) = \boldsymbol{\alpha}^\top \cdot \boldsymbol{\Delta}_w \cdot \boldsymbol{\gamma}$. We call $\Pr_{\mathcal{P}}(w)$ the *probability* of w in $\mathcal{P}$. Given a language $L \subseteq \Sigma^*$, we define the probability of L in P as $\Pr_{\mathcal{P}}(L) = \sum_{w \in L} \Pr_{\mathcal{P}}(w)$.

In some of the proofs later, we use the PA $\mathcal{P}_{Exp}$ defined as $\mathcal{P}_{Exp} = \big(\mathbf{1}, [\mu], \{[\mu]_a\}_{a\in\Sigma}\big)$ where $\mu = \frac{1}{|\Sigma|+1}$. $\mathcal{P}_{Exp}$ models a distribution over the words from Σ^* using a combination of an exponential distribution (for selecting the length l of a word) and the uniform distribution (for selecting symbols in a word of the length l). In particular, the purpose of $\mathcal{P}_{Exp}$ is to assign every word $w \in \Sigma^*$ the (nonzero) probability $\Pr_{\mathcal{P}_{Exp}}(w) = \mu^{|w|+1}$; any other PA assigning nonzero probabilities to all words would work as well.

If Conditions (i) and (ii) from the definition of PAs are dropped, we speak about a *pseudo-probabilistic automaton (PPA)*, which may assign a word from its support a quantity that is not necessarily in the range $\langle 0, 1\rangle$, denoted as the *significance* of the word below. PPAs may arise during some of our operations performed on PAs. Note that PPAs can be seen as instantiations of multiplicity or weighted automata [46].

3 Approximate reduction of NFAs

In this section, we first introduce the key notion of our approach: a *probabilistic distance* of a pair of finite automata wrt a given probabilistic automaton that, intuitively, represents the significance of particular words. We discuss the complexity of computing the probabilistic distance. Finally, we formulate two problems of *approximate automata reduction via probabilistic distance*.

3.1 Probabilistic distance

We start by defining our notion of a probabilistic distance of two NFAs. Assume NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ and a probabilistic automaton $\mathcal{P}$ specifying the distribution $\Pr_{\mathcal{P}} : \Sigma^* \to \langle 0, 1\rangle$. The *probabilistic distance* $d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2)$ between $\mathcal{A}_1$ and $\mathcal{A}_2$ wrt $\Pr_{\mathcal{P}}$ is defined as

$$d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2) = \Pr_{\mathcal{P}}(L(\mathcal{A}_1) \triangle L(\mathcal{A}_2)).$$

Intuitively, the distance captures the significance of the words accepted by one of the automata only. We use the distance to drive the reduction process towards automata with small errors and to assess the quality of the result. (The distance is sometimes called the *symmetric difference semi-metric* [17].)

The value of $\Pr_{\mathcal{P}}(L(\mathcal{A}_1) \triangle L(\mathcal{A}_2))$ can be computed as follows. Using the fact that (1) $L_1 \triangle L_2 = (L_1 \setminus L_2) \uplus (L_2 \setminus L_1)$ and (2) $L_1 \setminus L_2 = L_1 \setminus (L_1 \cap L_2)$, we get

$$\begin{aligned} &d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2)\\ &= \Pr_{\mathcal{P}}(L(\mathcal{A}_1)\setminus L(\mathcal{A}_2)) + \Pr_{\mathcal{P}}(L(\mathcal{A}_2)\setminus L(\mathcal{A}_1))\\ &= \Pr_{\mathcal{P}}(L(\mathcal{A}_1)\setminus(L(\mathcal{A}_1) \cap L(\mathcal{A}_2)))\\ &\quad + \Pr_{\mathcal{P}}(L(\mathcal{A}_2)\setminus(L(\mathcal{A}_2) \cap L(\mathcal{A}_1)))\\ &= \Pr_{\mathcal{P}}(L(\mathcal{A}_1)) + \Pr_{\mathcal{P}}(L(\mathcal{A}_2)) - 2 \cdot \Pr_{\mathcal{P}}(L(\mathcal{A}_1) \cap L(\mathcal{A}_2)). \end{aligned}$$

Hence, the key step is to compute $\Pr_{\mathcal{P}}(L(\mathcal{A}))$ for an NFA $\mathcal{A}$ and a PA $\mathcal{P}$. Problems similar to computing such a probability have been extensively studied in several contexts including verification of probabilistic systems [2,6,52].

In our approach, we apply the method of [6] and compute $\Pr_{\mathcal{P}}(L(\mathcal{A}))$ in the following way. We first check whether the NFA $\mathcal{A}$ is unambiguous. This can be done by using the standard product construction (denoted as $\cap$) for computing the intersection of the NFA $\mathcal{A}$ with itself and trimming the result, formally $\mathcal{B} = \mathit{trim}(\mathcal{A} \cap \mathcal{A})$, followed by a check whether there is some state $(p, q) \in Q[\mathcal{B}]$ s.t. $p \neq q$ [40]. If $\mathcal{A}$ is ambiguous, we either determinize it or disambiguate it [40], leading to a DFA/UFA $\mathcal{A}'$, respectively.[1] Then, we construct the trimmed product of $\mathcal{A}'$ and $\mathcal{P}$ (this can be seen as computing $\mathcal{A}' \cap \mathit{supp}(\mathcal{P})$ while keeping the probabilities from $\mathcal{P}$ on the edges of the result), yielding a PPA $\mathcal{R} = (\boldsymbol{\alpha}_{\mathcal{R}}, \boldsymbol{\gamma}_{\mathcal{R}}, \{\boldsymbol{\Delta}_a^{\mathcal{R}}\}_{a\in\Sigma})$.[2] Intuitively, $\mathcal{R}$ represents not only the words of $L(\mathcal{A})$ but also their probability in $\mathcal{P}$ (we give the formal definition of $\mathcal{R}$ inside the proof of Lemma 2). Now, let $\boldsymbol{\Delta} = \sum_{a\in\Sigma} \boldsymbol{\Delta}_a$ be the matrix that expresses, for any $p, q \in Q[\mathcal{R}]$, the significance of getting from p to q via any $a \in \Sigma$. Further, it can be shown (cf. the proof of Lemma 1) that the matrix $\boldsymbol{\Delta}^*$, representing the significance of going from p to q via any $w \in \Sigma^*$, can be computed as $(\boldsymbol{I} - \boldsymbol{\Delta})^{-1}$. Then, to get $\Pr_{\mathcal{P}}(L(\mathcal{A}))$, it suffices to take $\boldsymbol{\alpha}^\top \cdot \boldsymbol{\Delta}^* \cdot \boldsymbol{\gamma}$. Note that, due to the determinization/disambiguation step, the obtained value indeed is $\Pr_{\mathcal{P}}(L(\mathcal{A}))$ despite $\mathcal{R}$ being a PPA. The two lemmas below summarize the complexity of this step for NFAs and UFAs, respectively.

Lemma 1 *Let $\mathcal{P}$ be a PA and $\mathcal{A}$ an NFA. The problem of computing* $\Pr_{\mathcal{P}}(L(\mathcal{A}))$ *is* ***PSPACE****-complete.*

Proof The membership in **PSPACE** can be shown as follows. The computation described above corresponds to solving a linear equation system. The system has an exponential size because of the blowup caused by the determinization/disambiguation of $\mathcal{A}$ required by the product construction. The equation system can, however, be constructed by a **PSPACE** transducer $\mathcal{M}_{eq}$. Moreover, as solving linear equation systems can be done using a polylogarithmic-space transducer $\mathcal{M}_{SysLin}$, one can combine these two transducers to obtain a **PSPACE** algorithm. Details of the construction follow:

First, we construct a transducer $\mathcal{M}_{eq}$ that, given an NFA $\mathcal{A} = (Q_{\mathcal{A}}, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and a PA $\mathcal{P} = (\boldsymbol{\alpha}, \boldsymbol{\gamma}, \{\boldsymbol{\Delta}_a\}_{a\in\Sigma})$ on its input, constructs a system of $m = 2^{|Q_{\mathcal{A}}|} \cdot |Q[\mathcal{P}]|$ linear equations $\mathcal{S}(\mathcal{A}, \mathcal{P})$ of m unknowns $\xi_{[R,p]}$ for $R \subseteq Q_{\mathcal{A}}$ and $p \in Q[\mathcal{P}]$ representing the product of $\mathcal{A}'$ and $\mathcal{P}$, where $\mathcal{A}'$

[1] In theory, disambiguation can produce smaller automata, but, in our experiments, determinization proved to work better.

[2] $\mathcal{R}$ is not necessarily a PA since there might be transitions in $\mathcal{P}$ that are either removed or copied several times in the product construction.

is a deterministic automaton obtained from $\mathcal{A}$ using the standard subset construction. The system of equations $\mathcal{S}(\mathcal{A}, \mathcal{P})$ is defined as follows (cf. [6]):

$$\xi_{[R,p]} = \begin{cases} 0 & \text{if } L_{\mathcal{A}}(R) \cap L_{\mathcal{P}'}(p) = \emptyset, \\ \sum_{a\in\Sigma} \sum_{p' \in Q[\mathcal{P}]} \left(\boldsymbol{\Delta}_a[p,p'] \cdot \xi_{[\delta_{\mathcal{A}}(R,a),p']}\right) + \boldsymbol{\gamma}[p] & \\ & \text{if } R \cap F_{\mathcal{A}} \neq \emptyset, \\ \sum_{a\in\Sigma} \sum_{p' \in Q[\mathcal{P}]} \boldsymbol{\Delta}_a[p,p'] \cdot \xi_{[\delta_{\mathcal{A}}(R,a),p']} & \\ & \text{otherwise,} \end{cases}$$

such that $\mathcal{P}' = supp(\mathcal{P})$ and $\delta_{\mathcal{A}}(R, a) = \bigcup_{r\in R} \delta(r, a)$. The test $L_{\mathcal{A}}(R) \cap L_{\mathcal{P}'}(p) = \emptyset$ can be performed by checking $\exists r \in R : L_{\mathcal{A}}(r) \cap L_{\mathcal{P}'}(p) = \emptyset$, which can be done in polynomial time.

It holds that $\Pr_{\mathcal{P}}(L(\mathcal{A})) = \sum_{p\in Q[\mathcal{P}]} \boldsymbol{\alpha}[p] \cdot \xi_{[I_{\mathcal{A}},p]}$. Although the size of $\mathcal{S}(\mathcal{A}, \mathcal{P})$ (which is the output of $\mathcal{M}_{eq}$) is exponential in the size of the input of $\mathcal{M}_{eq}$, the internal configuration of $\mathcal{M}_{eq}$ only needs to be of polynomial size, i.e. $\mathcal{M}_{eq}$ works in **PSPACE**. Note that the size of each equation is at most polynomial.

Given a system $\mathcal{S}$ of m linear equations with m unknowns, solving $\mathcal{S}$ can be done in the time $\mathcal{O}(\log^2 m)$ using $\mathcal{O}(m^k)$ processors for a fixed k [16, Corollary 2] (i.e. it is in the class **NC**).[3] According to [19, Lemma 1b], an $\mathcal{O}(\log^2 m)$ time-bounded parallel machine can be simulated by an $\mathcal{O}(\log^4 m)$ space-bounded Turing machine. Therefore, there exists an $\mathcal{O}(\log^4 m)$ space-bounded Turing machine $\mathcal{M}_{SysLin}$ that solves a system of m linear equations with m unknowns. As a consequence, $\mathcal{M}_{SysLin}$ can solve $\mathcal{S}(\mathcal{A}, \mathcal{P})$ using the space

$$\begin{aligned} \mathcal{O}(\log^4(2^{|Q_{\mathcal{A}}|} \cdot |Q[\mathcal{P}]|)) &= \mathcal{O}(\log^4 2^{|Q_{\mathcal{A}}|} + \log^4 |Q[\mathcal{P}]|) \\ &= \mathcal{O}(|Q_{\mathcal{A}}|^4 + \log^4 |Q[\mathcal{P}]|). \end{aligned}$$

The missing part is how to combine $\mathcal{M}_{eq}$ and $\mathcal{M}_{SysLin}$ to avoid using the exponential-size output tape of $\mathcal{M}_{eq}$. For this, we use the following standard technique for combining reductions [43, Proposition 8.2].

We take turns in simulating $\mathcal{M}_{SysLin}$ and $\mathcal{M}_{eq}$. We start with simulating $\mathcal{M}_{SysLin}$. When $\mathcal{M}_{SysLin}$ moves its head right, we pause it and simulate $\mathcal{M}_{eq}$ until it outputs the corresponding bit, which is fed into the input of $\mathcal{M}_{SysLin}$. Then we pause $\mathcal{M}_{eq}$ and resume the run of $\mathcal{M}_{SysLin}$. On the other hand, when $\mathcal{M}_{SysLin}$ moves its head left (from the k-th position on the tape), we pause it, restart $\mathcal{M}_{eq}$ from its initial state, and simulate it until it outputs the $(k-1)$-st bit of its output tape, and then pause $\mathcal{M}_{eq}$ and return the control to $\mathcal{M}_{SysLin}$. In order to keep track of the position k of the head of $\mathcal{M}_{SysLin}$ on its tape, we use a binary counter.

[3] We use $\log k$ to denote the base-2 logarithm of k.

The internal configuration of both $\mathcal{M}_{eq}$ and $\mathcal{M}_{SysLin}$ is of a polynomial size and the overhead of keeping track of the position of the head of $\mathcal{M}_{SysLin}$ also requires only polynomial space. Therefore, the whole transducer runs in a polynomially bounded space.

The **PSPACE**-hardness is obtained by a reduction from the (**PSPACE**-complete) universality of NFAs: using the PA $\mathcal{P}_{Exp}$ defined in Sect. 2, which assigns every word a nonzero probability. it holds that

$$L(\mathcal{A}) = \Sigma^* \quad \text{iff} \quad \Pr_{\mathcal{P}_{Exp}}(L(\mathcal{A})) = 1.$$

□

Lemma 2 *Let $\mathcal{P}$ be a PA and $\mathcal{A}$ a UFA. The problem of computing $\Pr_{\mathcal{P}}(L(\mathcal{A}))$ is in* ***PTIME***.

Proof We modify the proof from [6] into our setting. First, we give a formal definition of the *product* of a PA $\mathcal{P} = (\boldsymbol{\alpha}, \boldsymbol{\gamma}, \{\boldsymbol{\Delta}_a\}_{a\in\Sigma})$ and an NFA $\mathcal{A} = (Q, \delta, I, F)$ as the $(|Q[\mathcal{P}]| \cdot |Q|)$-state PPA $\mathcal{R} = (\boldsymbol{\alpha}_{\mathcal{R}}, \boldsymbol{\gamma}_{\mathcal{R}}, \{\boldsymbol{\Delta}^{\mathcal{R}}_a\}_{a\in\Sigma})$ where[4]

$$\begin{aligned} \boldsymbol{\alpha}_{\mathcal{R}}[(q_{\mathcal{P}}, q_{\mathcal{A}})] &= \boldsymbol{\alpha}_{\mathcal{R}}[q_{\mathcal{P}}] \cdot |\{q_{\mathcal{A}}\} \cap I|, \\ \boldsymbol{\gamma}_{\mathcal{R}}[(q_{\mathcal{P}}, q_{\mathcal{A}})] &= \boldsymbol{\gamma}_{\mathcal{R}}[q_{\mathcal{P}}] \cdot |\{q_{\mathcal{A}}\} \cap F|, \\ \boldsymbol{\Delta}^{\mathcal{R}}_a[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] &= \boldsymbol{\Delta}_a[q_{\mathcal{P}}, q'_{\mathcal{P}}] \cdot |\{q'_{\mathcal{A}}\} \cap \delta(q_{\mathcal{A}}, a)|. \end{aligned}$$

Note that $\mathcal{R}$ is not necessarily a PA any more because for $w \in \Sigma^*$ such that $\Pr_{\mathcal{P}}(w) > 0$, (i) if $w \notin L(A)$, then $\Pr_{\mathcal{R}}(w) = 0$ and (ii) if $w \in L(\mathcal{A})$ and $\mathcal{A}$ can accept w using n different runs, then $\Pr_{\mathcal{R}}(w) = n \cdot \Pr_{\mathcal{P}}(w)$. As a consequence, the probabilities of all words from Σ^* are no longer guaranteed to add up to 1. If $\mathcal{A}$ is unambiguous, the second issue is avoided and $\mathcal{R}$ preserves the probabilities of words from $L(\mathcal{A})$, i.e. $\Pr_{\mathcal{R}}(w) = \Pr_{\mathcal{P}}(w)$ for all $w \in L(\mathcal{A})$, so $\mathcal{R}$ can be seen as the restriction of $\Pr_{\mathcal{P}}$ to $L(\mathcal{A})$. In the following, we assume $\mathcal{R}$ is trimmed.

In order to compute $\Pr_{\mathcal{P}}(L(\mathcal{A}))$, we construct a matrix $\boldsymbol{E}$ defined as $\boldsymbol{E} = \sum_{a\in\Sigma} \boldsymbol{\Delta}^{\mathcal{R}}_a$. Because $\mathcal{R}$ is trimmed, the *spectral radius* of $\boldsymbol{E}$, denoted as $\rho(\boldsymbol{E})$, is less than one, i.e. $\rho(\boldsymbol{E}) < 1$. (The proof of this fact can be found, for example, in [6].) Intuitively, $\rho(\boldsymbol{E}) < 1$ holds because we trimmed the redundant states from the product of $\mathcal{P}$ and $\mathcal{A}$. We further use the following standard result in linear algebra: if $\rho(\boldsymbol{E}) < 1$, then (i) the matrix $\boldsymbol{I} - \boldsymbol{E}$ is invertible and (ii) the sum of powers of $\boldsymbol{E}$, denoted as $\boldsymbol{E}^*$, can be computed as $\boldsymbol{E}^* = \sum_{i=0}^{\infty} \boldsymbol{E}^i = (\boldsymbol{I} - \boldsymbol{E})^{-1}$ [23]. Moreover, note that matrix inversion can be done in polynomial time [48].

$\boldsymbol{E}^*$ represents the reachability between nodes of $\mathcal{R}$, i.e. $\boldsymbol{E}^*[r, r']$ is the sum of significances of all (possibly infinitely

[4] We assume an implicit bijection between states of the product $\mathcal{R}$ and $\{1, \dots, |Q[\mathcal{R}]|\}$.

many) paths from r to r' in $\mathcal{R}$. When related to $\mathcal{P}$ and $\mathcal{A}$, the matrix $\boldsymbol{E}^*$ represents the reachability in $\mathcal{P}$ wrt $L(\mathcal{A})$, i.e.

$$\boldsymbol{E}^*[(q_\mathcal{P}, q_\mathcal{A}), (q'_\mathcal{P}, q'_\mathcal{A})] = \sum \left\{ \boldsymbol{\Delta}_w[q_\mathcal{P}, q'_\mathcal{P}] \mid q_\mathcal{A} \overset{w}{\rightsquigarrow} q'_\mathcal{A}, w \in \Sigma^* \right\}. \quad (1)$$

We prove Equation (1) using the following reasoning. First, we show that

$$\boldsymbol{E}^n[(q_\mathcal{P}, q_\mathcal{A}), (q'_\mathcal{P}, q'_\mathcal{A})] = \sum \left\{ \boldsymbol{\Delta}_w[q_\mathcal{P}, q'_\mathcal{P}] \mid q_\mathcal{A} \overset{w}{\rightsquigarrow} q'_\mathcal{A}, w \in \Sigma^n \right\}, \quad (2)$$

i.e. $\boldsymbol{E}^n$ represents the reachability in $\mathcal{P}$ wrt $L(\mathcal{A})$ for words of length n. We prove Equation (2) by induction on n: For $n = 0$, the equation follows from the fact that $\boldsymbol{E}^0 = \boldsymbol{I}$. For $n = 1$, the equation follows directly from the definition of $\mathcal{R}$ and $\boldsymbol{\Delta}$. Next, suppose that Equation (2) holds for $n > 1$; we show that it holds also for $n+1$. We start with the following reasoning:

$$\begin{aligned} bE^{n+1}&[(q_\mathcal{P}, q_\mathcal{A}), (q'_\mathcal{P}, q'_\mathcal{A})] \\ &= \boldsymbol{E}^n \boldsymbol{E})[(q_\mathcal{P}, q_\mathcal{A}), (q'_\mathcal{P}, q'_\mathcal{A})] \\ &= sum \Big\{ \boldsymbol{E}^n[(q_\mathcal{P}, q_\mathcal{A}), (q''_\mathcal{P}, q''_\mathcal{A})] \cdot \boldsymbol{E}[(q''_\mathcal{P}, q''_\mathcal{A}), \\ &\quad (q'_\mathcal{P}, q'_\mathcal{A})] \Big| (q''_\mathcal{P}, q''_\mathcal{A}) \in Q[\mathcal{R}] \Big\}. \end{aligned}$$

The last line is obtained via the definition of matrix multiplication. Further, using the induction hypothesis, we get

$$\begin{aligned} bE^{n+1}&[(q_\mathcal{P}, q_\mathcal{A}), (q'_\mathcal{P}, q'_\mathcal{A})] \\ &= sum \Big\{ \sum \Big\{ \boldsymbol{\Delta}_w[q_\mathcal{P}, q''_\mathcal{P}] \Big| q_\mathcal{A} \overset{w}{\rightsquigarrow} q''_\mathcal{A}, w \in \Sigma^n \Big\} \cdot \\ &\quad \sum \Big\{ \boldsymbol{\Delta}_a[q''_\mathcal{P}, q'_\mathcal{P}] \Big| q''_\mathcal{A} \xrightarrow{a} q'_\mathcal{A}, a \in \Sigma \Big\} \Big| \\ &\quad (q''_\mathcal{P}, q''_\mathcal{A}) \in Q[\mathcal{R}] \Big\} \\ &= \sum \Big\{ \sum \Big\{ \boldsymbol{\Delta}_w[q_\mathcal{P}, q''_\mathcal{P}] \cdot \boldsymbol{\Delta}_a[q''_\mathcal{P}, q'_\mathcal{P}] \Big| q_\mathcal{A} \overset{w}{\rightsquigarrow} q''_\mathcal{A}, \\ &\quad q''_\mathcal{A} \xrightarrow{a} q'_\mathcal{A}, a \in \Sigma, w \in \Sigma^n \Big\} \Big| (q''_\mathcal{P}, q''_\mathcal{A}) \in Q[\mathcal{R}] \Big\} \\ &= \sum \Big\{ \boldsymbol{\Delta}_{w'}[q_\mathcal{P}, q'_\mathcal{P}] \Big| q_\mathcal{A} \overset{w'}{\rightsquigarrow} q'_\mathcal{A}, w' \in \Sigma^{n+1} \Big\}. \end{aligned}$$

Since $\boldsymbol{E}^* = \sum_{i=0}^{\infty} \boldsymbol{E}^i$, Equation (1) follows. Using the matrix $\boldsymbol{E}^*$, it remains to compute $\Pr_\mathcal{P}(L(\mathcal{A}))$ as

$$\Pr_\mathcal{P}(L(\mathcal{A})) = \boldsymbol{\alpha}_\mathcal{R}^\top \cdot \boldsymbol{E}^* \cdot \boldsymbol{\gamma}_\mathcal{R}.$$

□

3.2 Automata reduction using probabilistic distance

We now exploit the probabilistic distance introduced above to formulate the task of approximate reduction of NFAs as two optimization problems. Given an NFA $\mathcal{A}$ and a PA $\mathcal{P}$ specifying the distribution $\Pr_\mathcal{P} : \Sigma^* \to \langle 0, 1 \rangle$, we define

- **size-driven reduction**: for $n \in \mathbb{N}$, find an NFA $\mathcal{A}'$ such that $|\mathcal{A}'| \leq n$ and the distance $d_\mathcal{P}(\mathcal{A}, \mathcal{A}')$ is minimal,
- **error-driven reduction**: for $\epsilon \in \langle 0, 1 \rangle$, find an NFA $\mathcal{A}'$ such that $d_\mathcal{P}(\mathcal{A}, \mathcal{A}') \leq \epsilon$ and the size $|\mathcal{A}'|$ is minimal.

The following lemma shows that the natural decision problem underlying both of the above optimization problems is **PSPACE**-complete, which matches the complexity of computing the probabilistic distance as well as that of the *exact* reduction of NFAs [26].

Lemma 3 *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \mathbb{N}$, and an error bound $\epsilon \in \langle 0, 1 \rangle$. It is **PSPACE**-complete to determine whether there exists an NFA $\mathcal{A}'$ with n states s.t. $d_\mathcal{P}(\mathcal{A}, \mathcal{A}') \leq \epsilon$.*

Proof Membership in **PSPACE**: We non-deterministically generate an automaton $\mathcal{A}'$ with n states and test (in **PSPACE**, as shown in Lemma 1) that $d_\mathcal{P}(\mathcal{A}, \mathcal{A}') \leq \epsilon$. This shows the problem is in **NPSPACE** = **PSPACE**.

PSPACE-hardness: We use a reduction from the problem of checking universality of an NFA $\mathcal{A} = (Q, \delta, I, F)$ over Σ, i.e. from checking whether $L(\mathcal{A}) = \Sigma^*$, which is **PSPACE**-complete. First, for a reason that will become clear later, we test if $\mathcal{A}$ accepts all words over Σ of length 0 and 1, which can be done in polynomial time. It holds that $L(\mathcal{A}) = \Sigma^*$ iff there is a 1-state NFA $\mathcal{A}'$ s.t. $d_{\mathcal{P}_{Exp}}(\mathcal{A}, \mathcal{A}') \leq 0$. ($\mathcal{P}_{Exp}$ is defined in Sect. 2.) The implication from left to right is clear: $\mathcal{A}'$ can be constructed as $\mathcal{A}' = (\{q\}, \{q \xrightarrow{a} q \mid a \in \Sigma\}, \{q\}, \{q\})$. To show the reverse implication, we note that we have tested that $\{\epsilon\} \cup \Sigma \subseteq L(\mathcal{A})$. Since the probability of any word from $\{\epsilon\} \cup \Sigma \subseteq L(\mathcal{A})$ in $\mathcal{P}_{Exp}$ is nonzero, the only 1-state NFA that processes those words with zero error is the NFA $\mathcal{A}'$ defined above. Because the language of $\mathcal{A}'$ is $L(\mathcal{A}') = \Sigma^*$, it holds that $d_{\mathcal{P}_{Exp}}(\mathcal{A}, \mathcal{A}') \leq 0$ iff $L(\mathcal{A}) = \Sigma^*$. □

The notions defined above do not distinguish between introducing a *false positive* ($\mathcal{A}'$ accepts a word $w \notin L(\mathcal{A})$) or a *false negative* ($\mathcal{A}'$ rejects a word $w \in L(\mathcal{A})$) answers. To this end, we define *over-approximating* and *under-approximating* reductions as reductions for which the conditions $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ and $L(\mathcal{A}) \supseteq L(\mathcal{A}')$ hold.

A naïve solution to the reductions would enumerate all NFAs $\mathcal{A}'$ of sizes from 0 up to k (resp. $|\mathcal{A}|$), for each of them compute $d_\mathcal{P}(\mathcal{A}, \mathcal{A}')$, and take an automaton with the smallest probabilistic distance (resp. a smallest one satisfying

Algorithm 1: A greedy size-driven reduction

```
Input  : NFA $\mathcal{A} = (Q, \delta, I, F)$, PA $\mathcal{P}$, $n \geq 1$
Output: NFA $\mathcal{A}'$, $\epsilon \in \mathbb{R}$ s.t. $|\mathcal{A}'| \leq n$ and $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}') \leq \epsilon$
1 $V \leftarrow \emptyset$;
2 for $q \in Q$ in the order $\preceq_{\mathcal{A}, label(\mathcal{A},\mathcal{P})}$ do
3     $V \leftarrow V \cup \{q\}$; $\mathcal{A}' \leftarrow reduce(\mathcal{A}, V)$;
4     if $|\mathcal{A}'| \leq n$ then break
5 return $\mathcal{A}'$, $\epsilon = error(\mathcal{A}, V, label(\mathcal{A}, \mathcal{P}))$;
```

the restriction on $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}')$). Obviously, this approach is computationally infeasible.

4 A heuristic approach to approximate reduction

In this section, we introduce two techniques for approximate reduction of NFAs that avoid the need to iterate over all automata of a certain size. The first approach is based on under-approximating the automata by removing states—we call it the *pruning reduction*—while the second approach is based on over-approximating the automata by adding self-loops to states and removing redundant states—we call it the *self-loop reduction*. Finding an optimal automaton using these reductions is also **PSPACE**-complete, but more amenable to heuristics like greedy algorithms. We start with introducing two high-level greedy algorithms, one for the size- and one for the error-driven reduction, and follow by showing their instantiations for the pruning and the self-loop reduction. A crucial role in the algorithms is played by a function that labels states of the automata by an estimate of the error that will be caused when some of the reductions is applied at a given state.

4.1 A general algorithm for size-driven reduction

Algorithm 1 shows a general greedy method for performing the size-driven reduction. In order to use the same high-level algorithm in both directions of reduction (over-/under-approximating), it is parameterized with the functions: *label*, *reduce*, and *error*. The real intricacy of the procedure is hidden inside these three functions. Intuitively, $label(\mathcal{A}, \mathcal{P})$ assigns every state of an NFA $\mathcal{A}$ an approximation of the error that will be caused wrt the PA $\mathcal{P}$ when a reduction is applied at this state, while the purpose of $reduce(\mathcal{A}, V)$ is to create a new NFA $\mathcal{A}'$ obtained from $\mathcal{A}$ by introducing some error at states from V.[5] Further, $error(\mathcal{A}, V, label(\mathcal{A}, \mathcal{P}))$ estimates the error introduced by the application of $reduce(\mathcal{A}, V)$, possibly in a more precise (and costly) way than by just summing the concerned error labels: Such a computation is possible outside of the main computation loop. We show instantiations of these functions later, when discussing the reductions used. Moreover, the algorithm is also parameterized with a total order $\preceq_{\mathcal{A}, label(\mathcal{A},\mathcal{P})}$ that defines which states of $\mathcal{A}$ are processed first and which are processed later. The ordering may take into account the pre-computed labelling. The algorithm accepts an NFA $\mathcal{A}$, a PA $\mathcal{P}$, and $n \in \mathbb{N}$ and outputs a pair consisting of an NFA $\mathcal{A}'$ of the size $|\mathcal{A}'| \leq n$ and an error bound ϵ such that $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}') \leq \epsilon$.

The main idea of the algorithm is that it creates a set V of states where an error is to be introduced. V is constructed by starting from an empty set and adding states to it in the order given by $\preceq_{\mathcal{A}, label(\mathcal{A},\mathcal{P})}$, until the size of the result of $reduce(\mathcal{A}, V)$ has reached the desired bound n (in our setting, *reduce* is always antitone, i.e. for $V \subseteq V'$, it holds that $|reduce(\mathcal{A}, V)| \geq |reduce(\mathcal{A}, V')|$). We now define the necessary condition for *label*, *reduce*, and *error* that makes Algorithm 1 correct.

Condition C1 *holds if for every NFA $\mathcal{A}$, PA $\mathcal{P}$, and a set $V \subseteq Q[\mathcal{A}]$, we have that*

(a) $error(\mathcal{A}, V, label(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, reduce(\mathcal{A}, V))$,
(b) $|reduce(\mathcal{A}, Q[\mathcal{A}])| \leq 1$, *and*
(c) $reduce(\mathcal{A}, \emptyset) = \mathcal{A}$.

C1(a) ensures that the error computed by the reduction algorithm indeed over-approximates the exact probabilistic distance, **C1**(b) is a boundary condition for the case when the reduction is applied at every state of $\mathcal{A}$, and **C1**(c) ensures that when no error is to be introduced at any state, we obtain the original automaton.

Lemma 4 *Algorithm 1 is correct if **C1** holds.*

Proof Follows straightforwardly from Condition **C1**. □

4.2 A general algorithm for error-driven reduction

In Algorithm 2, we provide a high-level method of computing the error-driven reduction. The algorithm is in many ways similar to Algorithm 1; it also computes a set of states V where an error is to be introduced, but an important difference is that we compute an approximation of the error in each step and only add q to V if it does not raise the error over the threshold ϵ. Note that the *error* does not need to be monotone, so it may be advantageous to traverse all states from Q and not terminate as soon as the threshold is reached. The correctness of Algorithm 2 also depends on **C1**.

Lemma 5 *Algorithm 2 is correct if **C1** holds.*

Proof Follows straightforwardly from Condition **C1**. □

[5] We emphasize that this does not mean that states from V will be simply removed from $\mathcal{A}$—the performed operation depends on the particular reduction.

Algorithm 2: A greedy error-driven reduction.

```
Input  : NFA $\mathcal{A} = (Q, \delta, I, F)$, PA $\mathcal{P}$, $\epsilon \in \langle 0, 1\rangle$
Output: NFA $\mathcal{A}'$ s.t. $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}') \leq \epsilon$
1 $\ell \leftarrow label(\mathcal{A}, P)$;
2 $V \leftarrow \emptyset$;
3 for $q \in Q$ in the order $\preceq_{\mathcal{A}, label(\mathcal{A}, \mathcal{P})}$ do
4     $e \leftarrow error(\mathcal{A}, V \cup \{q\}, \ell)$;
5     if $e \leq \epsilon$ then $V \leftarrow V \cup \{q\}$
6 return $\mathcal{A}' = reduce(\mathcal{A}, V)$;
```

4.3 Pruning reduction

The pruning reduction is based on identifying a set of states to be removed from an NFA $\mathcal{A}$, under-approximating the language of $\mathcal{A}$. In particular, for $\mathcal{A} = (Q, \delta, I, F)$, the pruning reduction finds a set $R \subseteq Q$ and restricts $\mathcal{A}$ to $Q \backslash R$, followed by removing useless states, to construct a reduced automaton $\mathcal{A}' = trim(\mathcal{A}_{|Q \backslash R})$. Note that the natural decision problem corresponding to this reduction is also **PSPACE**-complete.

Lemma 6 *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \mathbb{N}$, and an error bound $\epsilon \in \langle 0, 1\rangle$. It is **PSPACE**-complete to determine whether there exists a subset of states $R \subseteq Q[\mathcal{A}]$ of size $|R| = n$ such that $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}_{|R}) \leq \epsilon$.*

Proof Membership in **PSPACE**: We non-deterministically generate a subset R of $Q[\mathcal{A}]$ having n states and test (in **PSPACE**, as shown in Lemma 1) that $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}_{|R}) \leq \epsilon$. This shows the problem is in **NPSPACE** = **PSPACE**.

PSPACE-hardness: We use a reduction from the **PSPACE**-complete problem of checking universality of an NFA $\mathcal{A} = (Q, \delta, I, F)$ over Σ. Consider a symbol $x \notin \Sigma$. Let us construct an NFA $\mathcal{A}'$ over $\Sigma \cup \{x\}$ s.t. $L(\mathcal{A}') = x^*.L(\mathcal{A})$. $\mathcal{A}'$ is constructed by adding a fresh state q_{new} to $\mathcal{A}$ that can loop over x and make a transition to any initial state of $\mathcal{A}$ over x: $\mathcal{A}' = (Q \uplus \{q_{new}\}, \delta \cup \{q_{new} \xrightarrow{x} q \mid q \in I \cup \{q_{new}\}\}, I \cup \{q_{new}\}, F)$. We set $n = |\mathcal{A}'| + 1$. Further, we also construct an $(n+1)$-state NFA $\mathcal{B}$ accepting the language $x^n.\Sigma^*$ defined as $\mathcal{B} = (Q_{\mathcal{B}}, \delta_{\mathcal{B}}, \{q_1\}, \{q_{n+1}\})$ where $Q_{\mathcal{B}} = \{q_1, \ldots, q_{n+1}\}$ and $\delta_{\mathcal{B}} = \{q_i \xrightarrow{x} q_{i+1} \mid 1 \leq i \leq n\} \cup \{q_{n+1} \xrightarrow{a} q_{n+1} \mid a \in \Sigma\}$. Moreover, let $\mathcal{P}$ be a PA representing a distribution $\Pr_{\mathcal{P}}$ that is defined for each $w \in (\Sigma \cup \{x\})^*$ as

$$\Pr_{\mathcal{P}}(w) = \begin{cases} \mu^{|w'|+1} & \text{for } w = x^n.w', w' \in \Sigma^*, \\ & \quad \text{and } \mu = \frac{1}{|\Sigma|+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Note that $\Pr_{\mathcal{P}}(x^n.w) = \Pr_{\mathcal{P}_{Exp}}(w)$ for $w \in \Sigma^*$, and $\Pr_{\mathcal{P}}(u) = 0$ for $u \notin x^n.\Sigma^*$ ($\mathcal{P}$ can be easily constructed from $\mathcal{P}_{Exp}$.) Also note that $\mathcal{B}$ accepts exactly those words w such that $\Pr_{\mathcal{P}}(w) \neq 0$ and that $\Pr_{\mathcal{P}}(L(\mathcal{B})) = 1$. Using the automata defined above, we construct an NFA $\mathcal{C} = \mathcal{A}' \cup \mathcal{B}$ where the union of two NFAs is defined as $\mathcal{A}_1 \cup \mathcal{A}_2 = (Q[\mathcal{A}_1] \uplus Q[\mathcal{A}_2], \delta[\mathcal{A}_1] \uplus \delta[\mathcal{A}_2], I[\mathcal{A}_1] \uplus I[\mathcal{A}_2], F[\mathcal{A}_1] \uplus F[\mathcal{A}_2])$. NFA $\mathcal{C}$ has $2n$ states, the language of $\mathcal{C}$ is $L(\mathcal{C}) = x^*.L(\mathcal{A}) \cup x^n.\Sigma^*$ and its probability is $\Pr_{\mathcal{P}}(L(\mathcal{C})) = 1$.

The important property of $\mathcal{C}$ is that if there exists a set $R \subseteq Q[\mathcal{C}]$ of the size $|R| = n$ s.t. $d_{\mathcal{P}}(\mathcal{C}, \mathcal{C}_{|R}) \leq 0$, then $L(\mathcal{A}) = \Sigma^*$. The property holds because since $|Q[\mathcal{A}']| = n-1$, when we remove n states from $\mathcal{C}$, at least one state from $Q[\mathcal{B}]$ is removed, making the whole subautomaton of $\mathcal{C}$ corresponding to $\mathcal{B}$ useless, and, therefore, $L(\mathcal{C}_{|R}) \subseteq x^*.L(\mathcal{A})$. Because $d_{\mathcal{P}}(\mathcal{C}, \mathcal{C}_{|R}) \leq 0$, we know that $\Pr_{\mathcal{P}}(L(\mathcal{C}_{|R})) = 1$, so $x^n.\Sigma^* \subseteq x^*.L(\mathcal{A}) = L(\mathcal{C}_{|R})$ and, therefore, $L(\mathcal{A}) = \Sigma^*$. For the other direction, if $L(\mathcal{A}) = \Sigma^*$, then there exists a set $R \subseteq Q[\mathcal{A}'] \cup Q[\mathcal{B}]$ of the size $|R| = n$ s.t. $d_{\mathcal{P}}(\mathcal{C}, \mathcal{C}_{|R}) \leq 0$. (In particular, R can be such that $R \subseteq Q[\mathcal{B}]$.) □

Although Lemma 6 shows that the pruning reduction is as hard as a general reduction (cf. Lemma 3), the pruning reduction is more amenable to using heuristics like the greedy algorithms from Sects. 4.1 and 4.2. We instantiate *reduce*, *error*, and *label* in these high-level algorithms in the following way (the subscript p stands for *pruning*):

$$reduce_p(\mathcal{A}, V) = trim(\mathcal{A}_{|Q \backslash V}),$$
$$error_p(\mathcal{A}, V, \ell) = \min_{V' \in \lfloor V \rfloor_p} \sum \left\{ \ell(q) \mid q \in V' \right\},$$

where $\lfloor V \rfloor_p$ is defined in the rest of this paragraph: Because of the use of *trim* in $reduce_p$, for a pair of sets V, V' s.t. $V \subset V'$, it holds that $reduce_p(\mathcal{A}, V)$ may, in general, yield the same automaton as $reduce_p(\mathcal{A}, V')$. Therefore, in order to obtain a tight approximation, we wish to compute the least error that is obtained when removing the states in V. We define a partial order $\sqsubseteq_p$ on 2^Q as $V_1 \sqsubseteq_p V_2$ iff $reduce_p(\mathcal{A}, V_1) = reduce_p(A, V_2)$ and $V_1 \subseteq V_2$, and use $\lfloor V \rfloor_p$ to denote the set of minimal elements of the set of elements that are smaller than V (wrt $\sqsubseteq_p$). The value of the approximation $error_p(A, V, \ell)$ is therefore the minimum of the sum of errors over all sets from $\lfloor V \rfloor_p$.

Note that the size of $\lfloor V \rfloor_p$ can again be exponential, and thus we employ a greedy approach for guessing an optimal V'. Clearly, this cannot affect the soundness of the algorithm, but only decreases the precision of the bound on the distance. Our experiments indicate that for automata appearing in NIDSes, this simplification has typically only a negligible impact on the precision of the bounds.

For computing the state labelling, we provide the following three functions, which differ in the precision they provide and the difficulty of their computation (naturally, more precise labellings are harder to compute): $label^1_p$, $label^2_p$, and $label^3_p$. Given an NFA $\mathcal{A}$ and a PA $\mathcal{P}$, they generate the labellings ℓ^1_p, ℓ^2_p, and ℓ^3_p, respectively, defined as

$$\ell^1_p(q) = \sum \left\{ \Pr_{\mathcal{P}}(L^{\flat}_{\mathcal{A}}(q')) \mid q' \in reach(\{q\}) \cap F \right\},$$
$$\ell^2_p(q) = \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(F \cap reach(q))\right),$$
$$\ell^3_p(q) = \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\right).$$

A state label $\ell(q)$ approximates the error of the words removed from $L(\mathcal{A})$ when q is removed. More concretely, $\ell^1_p(q)$ is a rough estimate saying that the error can be bounded by the sum of probabilities of the banguages of all final states reachable from q. (In the worst case, all those final states might become unreachable.) Note that $\ell^1_p(q)$ (1) counts the error of a word accepted in two different final states of $reach(q)$ twice and (2) it also considers words that are accepted in some final state in $reach(q)$ without going through q. The labelling ℓ^2_p deals with (1) by computing the total probability of the banguage of the set of all final states reachable from q, and the labelling ℓ^3_p in addition also deals with (2) by only considering words that traverse through q. (They can, however, be accepted in some final state not in $reach(q)$ by a run completely disjoint from q and $reach(q) \cap F$, so even ℓ^3_p can still be imprecise.) Note that if $\mathcal{A}$ is unambiguous, then $\ell^1_p = \ell^2_p$.

Each state labelling is given as the probability (or the sum of probabilities in the case of ℓ^1_p) of the language related to q. Therefore, when computing the particular label of q, we first modify $\mathcal{A}$ to obtain $\mathcal{A}'$ accepting the language related to the labelling. Then, we compute the value of $\Pr_{\mathcal{P}}(L(\mathcal{A}'))$ using the algorithm from Sect. 3.1. Recall that this step is in general costly, due to the determinization/disambiguation of $\mathcal{A}'$. The key property of the labelling computation resides in the fact that if $\mathcal{A}$ is composed of several disjoint sub-automata, the automaton $\mathcal{A}'$ is typically much smaller than $\mathcal{A}$ and thus the computation of the label is considerably less demanding. Since the automata appearing in regex matching for NIDS are composed of the union of "tentacles", the particular $\mathcal{A}'$s are very small, which enables an efficient component-wise computation of the labels.

The following lemma states the correctness of using the pruning reduction as an instantiation of Algorithms 1 and 2 and also the relation among ℓ^1_p, ℓ^2_p, and ℓ^3_p.

Lemma 7 *For every $x \in \{1, 2, 3\}$, the functions $reduce_p$, $error_p$, and $label^x_p$ satisfy* **C1**. *Moreover, consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, and let $\ell^x_p = label^x_p(\mathcal{A}, \mathcal{P})$ for $x \in \{1, 2, 3\}$. Then, for each $q \in Q[\mathcal{A}]$, we have $\ell^1_p(q) \geq \ell^2_p(q) \geq \ell^3_p(q)$.*

Proof We start by proving the inequalities $\ell^1_p(q) \geq \ell^2_p(q) \geq \ell^3_p(q)$ for each $q \in Q[\mathcal{A}]$, which will then help us prove the first part of the lemma. The first inequality follows from the fact that if the banguages of reachable final states are not disjoint, in the case of ℓ^1_p, we may sum probabilities of the same words multiple times. The second inequality follows from the inclusion $L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q) \subseteq L^{\flat}_{\mathcal{A}}(F \cap reach(q))$.

Second, we prove that the functions $reduce_p$, $error_p$, and $label^x_p$ satisfy the properties of **C1**:

– **C1**(a): In order to show the inequality

$$error_p(\mathcal{A}, V, label^x_p(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, reduce_p(\mathcal{A}, V)),$$

we prove it for $\ell^3_p = label^3_p(\mathcal{A}, \mathcal{P})$; the rest follows from $\ell^1_p(q) \geq \ell^2_p(q) \geq \ell^3_p(q)$, which is proved above. Consider some set of states $V \subseteq Q[\mathcal{A}]$ and a set $V' \in \lfloor V \rfloor_p$ s.t. for any $V'' \in \lfloor V \rfloor_p$, it holds that $\sum\{\ell^3_p(q) \mid q \in V'\} \leq \sum\{\ell^3_p(q) \mid q \in V''\}$. We have

$$\begin{aligned}
&L(\mathcal{A}) \bigtriangleup L(reduce_p(\mathcal{A}, V)) \\
&\quad = L(\mathcal{A}) \bigtriangleup L(reduce_p(\mathcal{A}, V')) && \wr \text{def. of } \sqsubseteq_p \wr \\
&\quad = langof\mathcal{A} \setminus L(reduce_p(\mathcal{A}, V')) \\
&\quad \wr L(\mathcal{A}) \supseteq L(reduce_p(\mathcal{A}, V')) \wr \\
&\quad \subseteq \bigcup_{q \in V'} L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q). && \wr \text{def. of } reduce_p \wr
\end{aligned} \tag{4}$$

Finally, using (4), we obtain

$$\begin{aligned}
&d_{\mathcal{P}}(\mathcal{A}, reduce_p(\mathcal{A}, V)) \\
&\quad = \Pr_{\mathcal{P}}(L(\mathcal{A}) \bigtriangleup L(reduce_p(\mathcal{A}, V'))) \\
&\quad \wr \text{def. of } d_{\mathcal{P}} \wr \\
&\quad \leq \sum_{q \in V'} \Pr_{\mathcal{P}}(L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)) && \wr (4) \wr \\
&\quad = \sum\{\ell^3_p(q) \mid q \in V'\} && \wr \text{def. of } \ell^3_p \wr \\
&\quad = \min_{V'' \in \lfloor V \rfloor_p} \sum\{\ell^3_p(q) \mid q \in V''\} && \wr \text{def. of } V' \wr \\
&\quad = error_p(\mathcal{A}, V, \ell^3_p). && \wr \text{def. of } error_p \wr
\end{aligned}$$

– **C1**(b): $|reduce_p(\mathcal{A}, Q[\mathcal{A}])| \leq 1$ because

$$|reduce_p(\mathcal{A}, Q[\mathcal{A}])| = |trim(\mathcal{A}_{|\emptyset})| = 0.$$

– **C1**(c): $reduce_p(\mathcal{A}, \emptyset) = \mathcal{A}$ since

$$reduce_p(\mathcal{A}, \emptyset) = trim(\mathcal{A}_{|Q[\mathcal{A}]}) = \mathcal{A}.$$

(We assume that $\mathcal{A}$ is trimmed at the input.) □

4.4 Self-loop reduction

The main idea of the self-loop reduction is to over-approximate the language of $\mathcal{A}$ by adding self-loops over every symbol at selected states. This makes some states of $\mathcal{A}$ redundant, allowing them to be removed without introducing any more error. Given an NFA $\mathcal{A} = (Q, \delta, I, F)$, the self-loop reduction searches for a set of states $R \subseteq Q$, which will have self-loops added, and removes other transitions leading out of these states, making some states unreachable. The unreachable states are then removed.

Formally, let $sl(\mathcal{A}, R)$ be the NFA $(Q \cup \{s\}, \delta', I, F \cup \{s\})$ where $s \notin Q$ and the transition function δ' is defined such that $\delta'(s, a) = \{s\}$ and, for all states $p \in Q$ and symbols $a \in \Sigma$, $\delta'(p, a) = (\delta(p, a) \setminus R) \cup \{s\}$ if $\delta(p, a) \cap R \neq \emptyset$ and $\delta'(p, a) = \delta(p, a)$ otherwise. Similarly to the pruning reduction, the natural decision problem corresponding to the self-loop reduction is also **PSPACE**-complete.

Lemma 8 *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \mathbb{N}$, and an error bound $\epsilon \in \langle 0, 1 \rangle$. It is **PSPACE**-complete to determine whether there exists a subset of states $R \subseteq Q[\mathcal{A}]$ of size $|R| = n$ such that $d_{\mathcal{P}}(\mathcal{A}, sl(\mathcal{A}, R)) \leq \epsilon$.*

Proof Membership in **PSPACE** can be proved in the same way as in the proof of Lemma 6.

PSPACE-hardness: We reduce from the **PSPACE**-complete problem of checking universality of an NFA $\mathcal{A} = (Q, \delta, I, F)$. First, we check whether $I[\mathcal{A}] \neq \emptyset$. We have that $L(\mathcal{A}) = \Sigma^*$ iff there exists a set of states $R \subseteq Q$ of the size $|R| = |Q|$ such that $d_{\mathcal{P}_{Exp}}(\mathcal{A}, sl(\mathcal{A}, R)) \leq 0$. (Note that this means that a self-loop is added to every state of $\mathcal{A}$.) □

The required functions in the error- and size-driven reduction algorithms are instantiated in the following way (the subscript sl stands for *self-loop*):

$$reduce_{sl}(\mathcal{A}, V) = trim(sl(\mathcal{A}, V)),$$
$$error_{sl}(\mathcal{A}, V, \ell) = \sum \{\ell(q) \mid q \in \min(\lfloor V \rfloor_{sl})\},$$

where $\lfloor V \rfloor_{sl}$ is defined in a similar manner as $\lfloor V \rfloor_p$ in the previous section (using a partial order $\sqsubseteq_{sl}$ defined similarly to $\sqsubseteq_p$; the difference is that in this case, the order $\sqsubseteq_{sl}$ has a single minimal element, though).

The functions $label^1_{sl}$, $label^2_{sl}$, and $label^3_{sl}$ compute the state labellings ℓ^1_{sl}, ℓ^2_{sl}, and ℓ^3_{sl} for an NFA $\mathcal{A}$ and a PA $\mathcal{P}$, which are defined as follows:

$$\ell^1_{sl}(q) = weight_{\mathcal{P}}(L^{\flat}_{\mathcal{A}}(q)),$$
$$\ell^2_{sl}(q) = \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).\Sigma^*\right),$$
$$\ell^3_{sl}(q) = \ell^2_{sl}(q) - \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\right).$$

In the definitions above, the function $weight_{\mathcal{P}}(w)$ for a PA $\mathcal{P} = (\boldsymbol{\alpha}, \boldsymbol{\gamma}, \{\boldsymbol{\Delta}_a\}_{a \in \Sigma})$ and a word $w \in \Sigma^*$ is defined as $weight_{\mathcal{P}}(w) = \boldsymbol{\alpha}^\top \cdot \boldsymbol{\Delta}_w \cdot \mathbf{1}$ (i.e. similarly as $\Pr_{\mathcal{P}}(w)$ but with the final weights $\boldsymbol{\gamma}$ discarded), and $weight_{\mathcal{P}}(L)$ for $L \subseteq \Sigma^*$ is defined as $weight_{\mathcal{P}}(L) = \sum_{w \in L} weight_{\mathcal{P}}(w)$.

Intuitively, the state labelling $\ell^1_{sl}(q)$ computes the probability that q is reached from an initial state, so if q is pumped up with all possible word endings, this is the maximum possible error introduced by the added word endings. This has the following sources of imprecision: (1) the probability of some words may be included twice, e.g. when $L^{\flat}_{\mathcal{A}}(q) = \{a, ab\}$, the probabilities of all words from $\{ab\}.\Sigma^*$ are included twice in $\ell^1_{sl}(q)$ because $\{ab\}.\Sigma^* \subseteq \{a\}.\Sigma^*$, and (2) $\ell^1_{sl}(q)$ can also contain probabilities of words already accepted on a run traversing q. The state labelling ℓ^2_{sl} deals with (1) by considering the probability of the language $L^{\flat}_{\mathcal{A}}(q).\Sigma^*$, and ℓ^3_{sl} deals also with (2) by subtracting from the result of ℓ^2_{sl} the probabilities of the words that pass through q and are accepted.

The computation of the state labellings for the self-loop reduction is done in a similar way as the computation of the state labellings for the pruning reduction (cf. Sect. 4.3). For a computation of $weight_{\mathcal{P}}(L)$, one can use the same algorithm as for $\Pr_{\mathcal{P}}(L)$, only the final vector for PA $\mathcal{P}$ is set to $\mathbf{1}$. The correctness of Algorithms 1 and 2 when instantiated using the self-loop reduction is stated in the following lemma.

Lemma 9 *For every $x \in \{1, 2, 3\}$, the functions $reduce_{sl}$, $error_{sl}$, and $label^x_{sl}$ satisfy **C1**. Moreover, consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, and let $\ell^x_{sl} = label^x_{sl}(\mathcal{A}, \mathcal{P})$ for $x \in \{1, 2, 3\}$. Then, for each $q \in Q[\mathcal{A}]$, we have $\ell^1_{sl}(q) \geq \ell^2_{sl}(q) \geq \ell^3_{sl}(q)$.*

Proof First, we prove the inequalities $\ell^1_{sl}(q) \geq \ell^2_{sl}(q) \geq \ell^3_{sl}(q)$ for each $q \in Q[\mathcal{A}]$, which we then use to prove the first part of the lemma. We start with the equality $weight_{\mathcal{P}}(w) = \Pr_{\mathcal{P}}(w.\Sigma^*)$, which follows from the fact that for each state p of $\mathcal{P}$ the sum of probabilities of all words, when considering p as the only initial state of $\mathcal{P}$, is 1. Then, we obtain the equality

$$\sum_{w \in L^{\flat}_{\mathcal{A}}(q)} weight_{\mathcal{P}}(w) = \sum_{w \in L^{\flat}_{\mathcal{A}}(q)} \Pr_{\mathcal{P}}(w.\Sigma^*),$$

which, in turn, implies

$$\begin{aligned} \ell^1_{sl}(q) = weight_{\mathcal{P}}(L^{\flat}_{\mathcal{A}}(q)) &= \sum_{w \in L^{\flat}_{\mathcal{A}}(q)} \Pr_{\mathcal{P}}\left(w.\Sigma^*\right) \\ &\geq \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).\Sigma^*\right) = \ell^2_{sl}(q). \end{aligned} \tag{5}$$

For example, for $L^{\flat}_{\mathcal{A}}(q) = \{w, wa\}$ where $w \in \Sigma^*$ and $a \in \Sigma$, we have

$$\begin{aligned} weight_{\mathcal{P}}(L^{\flat}_{\mathcal{A}}(q)) &= weight_{\mathcal{P}}(\{w, wa\}) \\ &= weight_{\mathcal{P}}(w) + weight_{\mathcal{P}}(wa) \\ &= \Pr_{\mathcal{P}}(w.\Sigma^*) + \Pr_{\mathcal{P}}(wa.\Sigma^*), \end{aligned} \tag{6}$$

while

$$\Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).\Sigma^*\right) = \Pr_{\mathcal{P}}\left(\{w, wa\}.\Sigma^*\right) = \Pr_{\mathcal{P}}\left(w.\Sigma^*\right).$$

The inequality $\ell^2_{sl} \geq \ell^3_{sl}$ holds trivially.

Second, we prove that the functions $reduce_{sl}$, $error_{sl}$, and $label^x_{sl}$ satisfy the properties of **C1**:

- **C1**(a): To show that $error_{sl}(\mathcal{A}, V, label^x_{sl}(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, reduce_{sl}(\mathcal{A}, V))$, we prove that the inequality holds for $\ell^3_{sl} = label^3_{sl}(\mathcal{A}, \mathcal{P})$; the rest follows from $\ell^1_{sl}(q) \geq \ell^2_{sl}(q) \geq \ell^3_{sl}(q)$ proved above.
  Consider some set of states $V \subseteq Q[\mathcal{A}]$ and the set $V' = \min(\lfloor V \rfloor_{sl})$. We can estimate the symmetric difference of the languages of the original and the reduced automaton as

$$\begin{aligned} &L(\mathcal{A}) \triangle L(reduce_{sl}(\mathcal{A}, V)) \\ &\quad = L(\mathcal{A}) \triangle L(reduce_{sl}(\mathcal{A}, V')) \qquad \wr\text{def. of } \sqsubseteq_{sl}\wr \\ &\quad = L(reduce_{sl}(\mathcal{A}, V')) \setminus L(\mathcal{A}) \\ &\qquad \wr L(\mathcal{A}) \subseteq L(reduce_{sl}(\mathcal{A}, V'))\wr \\ &\quad \subseteq \bigcup_{q \in V'} L^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \bigcup_{q \in V'} L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q). \\ &\qquad \wr\text{def. of } reduce_{sl}\wr \end{aligned} \tag{7}$$

  The last inclusion holds because $sl(\mathcal{A}, V)$ adds self-loops to the states in V, so the newly accepted words are for sure those that traverse through V, and they are for sure not those that could be accepted by going through V before the reduction (but they could be accepted without touching V, hence the inclusion). We can estimate the probabilistic distance of $\mathcal{A}$ and $reduce_{sl}(\mathcal{A}, V)$ as

$$\begin{aligned} &d_{\mathcal{P}}(\mathcal{A}, reduce_{sl}(\mathcal{A}, V)) \\ &\leq \Pr_{\mathcal{P}}\Big(\bigcup_{q \in V'} L^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \bigcup_{q \in V'} L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\Big) \qquad \wr(7)\wr \\ &\leq \Pr_{\mathcal{P}}\Big(\bigcup_{q \in V'} \left(L^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\right)\Big) \\ &\quad \wr\text{properties of union and set difference}\wr \\ &\leq \sum_{q \in V'} \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\right) \\ &\quad \wr\text{union bound}\wr \\ &= \sum_{q \in V'} \left(\Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).\Sigma^*\right) - \Pr_{\mathcal{P}}\left(L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q)\right)\right) \\ &\quad \wr\text{prop. of Pr and the fact that } L^{\flat}_{\mathcal{A}}(q).L_{\mathcal{A}}(q) \subseteq L^{\flat}_{\mathcal{A}}(q).\Sigma^*\wr \\ &= \sum\{\ell^3_{sl}(q) \mid q \in \min(\lfloor V \rfloor_{sl})\} \\ &\quad \wr\text{def. of } \ell^3_{sl} \text{ and } V'\wr \\ &= error_{sl}(\mathcal{A}, V, \ell^3_{sl}). \end{aligned} \tag{8}$$

- **C1**(b): $|reduce_{sl}(\mathcal{A}, Q[\mathcal{A}])| \leq 1$ because, from the definition, $|reduce_{sl}(\mathcal{A}, Q[\mathcal{A}])| = |trim(sl(\mathcal{A}, Q[\mathcal{A}]))| \leq 1$.
- **C1**(c): $reduce_{sl}(\mathcal{A}, \emptyset) = \mathcal{A}$ since

$$reduce_{sl}(\mathcal{A}, \emptyset) = trim(sl(\mathcal{A}, \emptyset)) = \mathcal{A}.$$

  (We assume that $\mathcal{A}$ is trimmed at the input.) □

5 Reduction of NFAs in network intrusion detection systems

We have implemented our approach in a Python prototype named APPREAL (APProximate REduction of Automata and Languages)[6] and evaluated it on the use case of network intrusion detection using SNORT [50], a popular open-source NIDS. The version of APPREAL used for the evaluation in the current paper is available as an artefact [11] for the TACAS'18 artefact virtual machine [22].

5.1 Network traffic model

The reduction we describe in this paper is driven by a probabilistic model representing a distribution over the words from Σ^*, and the formal guarantees are also wrt this model. We use *learning* to obtain a model of network traffic over the 8-bit ASCII alphabet at a given network point. Our model is created from several gigabytes of network traffic from a measuring point of the CESNET Internet provider connected to a 100 Gbps backbone link. (Unfortunately, we cannot provide the traffic dump since it may contain sensitive data.)

Learning a PA representing the network traffic faithfully is hard. The PA cannot be too specific—although the number of different packets that can occur is finite, it is still extremely large. (A conservative estimate assuming the most common scenario Ethernet/IPv4/TCP would still yield a number over $2^{10{,}000}$.) If we assigned nonzero probabilities only to the packets from the dump (which are less than 2^{20}), the obtained model would completely ignore virtually all packets that might appear on the network, and, moreover, the model would also be very large (millions of states), making it difficult to use in our algorithms. A generalization of the obtained traffic is therefore needed.

A natural solution is to exploit results from the area of PA learning, such as [10,51]. Indeed, we experimented with

[6] https://github.com/vhavlena/appreal/tree/tacas18

the use of ALERGIA [10], a learning algorithm that constructs a PA from a prefix tree (where edges are labelled with multiplicities) by merging nodes that are "similar." The automata that we obtained were, however, *too* general. In particular, the constructed automata destroyed the structure of network protocols—the merging was too permissive and the generalization merged distant states, which introduced loops over a very large substructure in the automaton. (Such a case usually does not correspond to the design of network protocols.) As a result, the obtained PA more or less represented the Poisson distribution, having essentially no value for us.

In Sect. 5.2, we focus on the detection of malicious traffic transmitted over HTTP. We take advantage of this fact and create a PA representing the traffic while taking into account the structure of HTTP. We start by manually creating a DFA that represents the high-level structure of HTTP. Then, we proceed by feeding 34,191 HTTP packets from our sample into the DFA, at the same time taking notes about how many times every state is reached and how many times every transition is taken. The resulting PA $\mathcal{P}_{HTTP}$ (of 52 states) is then constructed from the DFA and the labels in the obvious way.

The described method yields automata that are much better than those obtained using ALERGIA in our experiments. A disadvantage of the method is that it is only semi-automatic—the basic DFA needed to be provided by an expert. We have yet to find an algorithm that would suit our needs for learning more general network traffic.

5.2 Evaluation

We start this section by introducing the experimental setting, namely, the integration of our reduction techniques into the tool chain implementing efficient regex matching, the concrete settings of APPREAL, and the evaluation environment. Afterwards, we discuss the results evaluating the quality of the obtained approximate reductions as well as of the provided error bounds. Finally, we present the performance of our approach and discuss its key aspects. We selected the most interesting results demonstrating the potential as well as the limitations of our approach.

General setting. SNORT detects malicious network traffic based on *rules* that contain *conditions*. The conditions take into consideration, among others, network addresses, ports, or Perl compatible regular expressions (PCREs) that the packet payload should match. In our evaluation, we select a subset of SNORT rules, extract the PCREs from them, and use NETBENCH [45] to transform them into a single NFA $\mathcal{A}$. Before applying APPREAL, we use the state-of-the-art NFA reduction tool REDUCE [38] to reduce $\mathcal{A}$. REDUCE performs a language-preserving reduction of $\mathcal{A}$ using advanced variants of simulation [37]. (In the experiment reported in Table 3, we skip the use of REDUCE at this step as discussed later in the performance evaluation.) The automaton $\mathcal{A}^{\text{RED}}$ obtained as the result of REDUCE is the input of APPREAL, which performs one of the approximate reductions from Sect. 4 wrt the traffic model $\mathcal{P}_{HTTP}$, yielding $\mathcal{A}^{\text{APP}}$. After the approximate reduction, we, one more time, use REDUCE and obtain the result $\mathcal{A}'$.

Settings of APPREAL In the use case of NIDS pre-filtering, it may be important to never introduce a false negative, i.e. to never drop a malicious packet. Therefore, we focus our evaluation on the *self-loop reduction* (Sect. 4.4). In particular, we use the state labelling function $label^2_{sl}$, since it provides a good trade-off between the precision and the computational demands. (Recall that the computation of $label^2_{sl}$ can exploit the "tentacle" structure of the NFAs we work with.) We give more attention to the *size-driven reduction* (Sect. 4.1) since, in our setting, a bound on the available FPGA resources is typically given and the task is to create an NFA with the smallest error that fits inside. The order $\preceq_{\mathcal{A},\ell^2_{sl}}$ over states used in Sect. 4.1 and Sect. 4.2 is defined as $s \preceq_{\mathcal{A},\ell^2_{sl}} s' \Leftrightarrow \ell^2_{sl}(s) \leq \ell^2_{sl}(s')$.

Evaluation environment All experiments ran on a 64-bit LINUX DEBIAN workstation with the Intel Core(TM) i5-661 CPU running at 3.33 GHz with 16 GiB of RAM.

Description of tables In the caption of every table, we provide the name of the input file (in the directory `regexps/tacas18/` of the repository of APPREAL) with the selection of SNORT regexes used in the particular experiment, together with the type of the reduction (size- or error-driven). All reductions are over-approximating (self-loop reduction). We further provide the size of the input automaton $|\mathcal{A}|$, the size after the initial processing by REDUCE ($|\mathcal{A}^{\text{RED}}|$), and the time of this reduction (*time*(REDUCE)). Finally, we list the times of computing the state labelling $label^2_{sl}$ on $\mathcal{A}^{\text{RED}}$ (*time*($label^2_{sl}$)), the exact probabilistic distance (*time*(Exact)), and also the number of *look-up tables* (*LUTs*($\mathcal{A}^{\text{RED}}$)) consumed on the targeted FPGA (Xilinx Virtex 7 H580T) when $\mathcal{A}^{\text{RED}}$ was synthesized (more on this in Sect. 5.3). The meaning of the columns in the tables is the following:

$\boldsymbol{k/\epsilon}$	is the parameter of the reduction. In particular, $\boldsymbol{k}$ is used for the size-driven reduction and denotes the desired reduction ratio $k = \frac{n}{\lvert\mathcal{A}^{\text{RED}}\rvert}$ for an input NFA $\mathcal{A}^{\text{RED}}$ and the desired size of the output n. On the other hand, $\boldsymbol{\epsilon}$ is the desired maximum error on the output for the error-driven reduction.
$\lvert\boldsymbol{\mathcal{A}}^{\text{APP}}\rvert$	shows the number of states of the automaton $\mathcal{A}^{\text{APP}}$ after the reduction by APPREAL and the time the reduction took. (We omit it when it is not interesting.)

Table 1 Results for the `http-malicious` regex, $|\mathcal{A}_{\texttt{mal}}| = 249$, $|\mathcal{A}^{\text{RED}}_{\texttt{mal}}| = 98$, *time*(REDUCE) $= 3.5$ s, $time(label^2_{sl}) = 38.7$ s, *time*(Exact) $=$ 3.8–6.5 s, and $LUTs(\mathcal{A}^{\text{RED}}_{\texttt{mal}}) = 382$

k	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{mal}}\rvert$	$\lvert\mathcal{A}'_{\texttt{mal}}\rvert$	**Error bound**	**Exact error**	**Traffic error**	**LUTs**
(a) Size-driven reduction						
0.1	9 (0.65 s)	9 (0.4 s)	0.0704	0.0704	0.0685	–
0.2	19 (0.66 s)	19 (0.5 s)	0.0677	0.0677	0.0648	–
0.3	29 (0.69 s)	26 (0.9 s)	0.0279	0.0278	0.0598	154
0.4	39 (0.68 s)	36 (1.1 s)	0.0032	0.0032	0.0008	–
0.5	49 (0.68 s)	44 (1.4 s)	2.8e−05	2.8e−05	4.1e−06	–
0.6	58 (0.69 s)	49 (1.7 s)	8.7e−08	8.7e−08	0.0	224
0.8	78 (0.69 s)	75 (2.7 s)	2.4e−17	2.4e−17	0.0	297

ϵ	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{mal}}\rvert$	$\lvert\mathcal{A}'_{\texttt{mal}}\rvert$	**Error bound**	**Exact error**	**Traffic error**
(b) Error-driven reduction					
0.08	3	3	0.0724	0.0724	0.0720
0.07	4	4	0.0700	0.0700	0.0683
0.04	35	32	0.0267	0.0212	0.0036
0.02	36	33	0.0105	0.0096	0.0032
0.001	41	38	0.0005	0.0005	0.0003
1e−04	47	41	7.7e−05	7.7e−05	1.2e−05
1e−05	51	47	6.6e−06	6.6e−06	0.0

$\lvert\mathcal{A}'\rvert$	contains the number of states of the NFA $\mathcal{A}'$ obtained after applying REDUCE on $\mathcal{A}^{\text{APP}}$ and the time used by REDUCE at this step (omitted when not interesting).
Error bound	shows the estimation of the error of $\mathcal{A}'$ as determined by the reduction itself, i.e. it is the probabilistic distance computed by the corresponding function *error* from Sect. 4.
Exact error	contains the values of $d_{\mathcal{P}_{HTTP}}(\mathcal{A}, \mathcal{A}')$ that we computed *after* the reduction in order to evaluate the precision of the result given in **Error bound**. The computation of this value is very expensive ($time(Exact)$) since it inherently requires determinization of the whole automaton $\mathcal{A}$. We do not provide it in Table 3 (presenting the results for the automaton $\mathcal{A}_{\texttt{bd}}$ with 1,352 states) because the determinization ran out of memory. (The step is not required in the reduction process.)
Traffic error	shows the error that we obtained when compared $\mathcal{A}'$ with $\mathcal{A}$ on an HTTP traffic sample, in particular the ratio of packets misclassified by $\mathcal{A}'$ to the total number of packets in the sample (242,468). Comparing **Exact error** with **Traffic error** gives us a feedback about the fidelity of the traffic model $\mathcal{P}_{HTTP}$. We note that there are no guarantees on the relationship between **Exact error** and **Traffic error**.
LUTs	is the number of LUTs consumed by $\mathcal{A}'$ when synthesized into the target FPGA. Hardware synthesis is a costly step, therefore we provide this value only for selected interesting NFAs.

5.2.1 Approximation errors

Table 1 presents the results of the self-loop reduction for the NFA $\mathcal{A}_{\texttt{mal}}$ describing regexes from `http-malicious`. We can observe that the differences between the upper bounds on the probabilistic distance and its real value are negligible (typically in the order of 10^{-4} or less). We can also see that the probabilistic distance agrees with the traffic error. This indicates a good quality of the traffic model employed in the reduction process. Further, we can see that our approach can provide useful trade-offs between the reduction error and the reduction factor. Finally, Table 1b shows that a significant reduction is obtained when the error threshold ϵ is increased from 0.04 to 0.07.

Table 2 presents the results of the size-driven self-loop reduction for NFA $\mathcal{A}_{\texttt{att}}$ describing `http-attacks` regexes. We can observe that the error bounds provide again a very good approximation of the real probabilistic distance.

Table 2 Results for the `http-attacks` regex, size-driven reduction, $|\mathcal{A}_{\texttt{att}}| = 142$, $|\mathcal{A}^{\textsc{Red}}_{\texttt{att}}| = 112$, *time*(REDUCE) $= 7.9$ s, *time*($label^2_{sl}$) $=$ 28.3 min, *time*(Exact) $=$ 14.0–16.4 min

k	$\lvert\mathcal{A}^{\textsc{App}}_{\texttt{att}}\rvert$	$\lvert\mathcal{A}'_{\texttt{att}}\rvert$	Error bound	Exact error	Traffic error
0.1	11 (1.1s)	5 (0.4s)	1.0	0.9972	0.9957
0.2	22 (1.1s)	14 (0.6s)	1.0	0.8341	0.2313
0.3	33 (1.1s)	24 (0.7s)	0.081	0.0770	0.0067
0.4	44 (1.1s)	37 (1.6s)	0.0005	0.0005	0.0010
0.5	56 (1.1s)	49 (1.2s)	3.3e−06	3.3e−06	0.0010
0.6	67 (1.1s)	61 (1.9s)	1.2e−09	1.2e−09	8.7e−05
0.7	78 (1.1s)	72 (2.4s)	4.8e−12	4.8e−12	1.2e−05
0.9	100 (1.1s)	93 (4.7s)	3.7e−16	1.1e−15	0.0

Table 3 Results for `http-backdoor`, size-driven reduction, $|\mathcal{A}_{\texttt{bd}}| = 1,352$, *time*($label^2_{sl}$) $= 19.9$ min, *LUTs*($\mathcal{A}^{\textsc{Red}}_{\texttt{bd}}$) $= 2,266$

k	$\lvert\mathcal{A}^{\textsc{App}}_{\texttt{bd}}\rvert$	$\lvert\mathcal{A}'_{\texttt{bd}}\rvert$	Error bound	Traffic error	LUTs
0.1	135 (1.2 m)	8 (2.6 s)	1.0	0.997	202
0.2	270 (1.2 m)	111 (5.2 s)	0.0012	0.0631	579
0.3	405 (1.2 m)	233 (9.8 s)	3.4e−08	0.0003	894
0.4	540 (1.3 m)	351 (21.7 s)	1.0e−12	0.0003	1,063
0.5	676 (1.3 m)	473 (41.8 s)	1.2e−17	0.0	1,249
0.7	946 (1.4 m)	739 (2.1 m)	8.3e−30	0.0	1,735
0.9	1216 (1.5 m)	983 (5.6 m)	1.3e−52	0.0	2,033

On the other hand, the difference between the probabilistic distance and the traffic error is larger than that for $\mathcal{A}_{\texttt{mal}}$. Since all experiments use the same probabilistic automaton and the same traffic, this discrepancy is accounted to the different set of packets that are incorrectly accepted by $\mathcal{A}^{\textsc{Red}}_{\texttt{att}}$. If the probability of these packets is adequately captured in the traffic model, the difference between the distance and the traffic error is small and vice versa. This also explains an even larger difference in Table 3 (presenting the results for $\mathcal{A}_{\texttt{bd}}$ constructed from `http-backdoor` regexes) for $k \in \langle 0.2, 0.4\rangle$. Here, the traffic error is very small and caused by a small set of packets (approx. 70), whose probability is not correctly captured in the traffic model. Despite this problem, the results clearly show that our approach still provides significant reductions while keeping the traffic error small: about a fivefold reduction is obtained for the traffic error 0.03 % and a tenfold reduction is obtained for the traffic error 6.3 %. We discuss the practical impact of such a reduction in Sect. 5.3.

5.2.2 Performance of the approximate reduction

In all our experiments (Tables 1, 2, 3), we can observe that the most time-consuming step of the reduction process is the computation of state labellings. (It takes at least 90 % of the total time.) The crucial observation is that the structure of the NFAs fundamentally affects the performance of this step. Although after REDUCE, the size of $\mathcal{A}_{\texttt{mal}}$ is very similar to the size of $\mathcal{A}_{\texttt{att}}$, computing $label^2_{sl}$ takes more time (28.3 min vs. 38.7 s). The key reason behind this slowdown is the determinization (or alternatively disambiguation) process required by the product construction underlying the state labelling computation (cf. Sect. 4.4). For $\mathcal{A}_{\texttt{att}}$, the process results in a significantly larger product when compared to the product for $\mathcal{A}_{\texttt{mal}}$. The size of the product directly determines the time and space complexity of solving the linear equation system required for computing the state labelling.

As explained in Sect. 4, the computation of the state labelling $label^2_{sl}$ can exploit the “tentacle” structure of the NFAs appearing in NIDSes and thus can be done component-wise. On the other hand, our experiments reveal that the use of REDUCE typically breaks this structure and thus the component-wise computation cannot be effectively used. For the NFA $\mathcal{A}_{\texttt{mal}}$, this behaviour does not have any major performance impact as the determinization leads to a moderate-sized automaton and the state labelling computation takes less than 40 s. On the other hand, this behaviour has a dramatic effect for the NFA $\mathcal{A}_{\texttt{att}}$. By disabling the initial application of REDUCE and thus preserving the original structure of $\mathcal{A}_{\texttt{att}}$, we were able to speed up the state label computation from 28.3 to 1.5 min. Note that other steps of the approximate reduction took a similar time as before disabling REDUCE and also that the trade-offs between the error and the reduction factor were similar. Surprisingly, disabling REDUCE caused that the computation of the exact probabilistic distance became computationally infeasible because the determinization ran out of memory.

Due to the size of the NFA $\mathcal{A}_{\texttt{bd}}$, the impact of disabling the initial application of REDUCE is even more fundamental. In particular, computing the state labelling took only 19.9 min, in contrast to running out of memory when the REDUCE is applied in the first step. (Therefore, the input automaton is not processed by REDUCE in Table 3; we still give the number of LUTs of its reduced version for comparison, though.) Note that the size of $\mathcal{A}_{\texttt{bd}}$ also slows down other reduction steps (the greedy algorithm and the final REDUCE reduction). We can, however, clearly see that computing the state labelling is still the most time-consuming step of the process.

5.3 The real impact in an FPGA-accelerated NIDS

To demonstrate the practical usefulness and impact of the proposed approximation techniques, we employ the reduced automata in a real use case from the area of HW-accelerated deep packet inspection. We consider the framework of [36] implementing a high-speed NIDS pre-filter in an FPGA. The crucial challenge is to obtain a pre-filter with a sufficiently small false positive rate (and no false negatives), while being able to handle the traffic of current networks operating on 100 Gbps and beyond. The implementation of NFAs performing regex matching in FPGAs uses two types of HW resources: LUTs, which are used to build the combinational circuit representing the NFA transition function, and flip-flops, representing NFA states. In our use case, we omit the analysis of flip-flop consumption because it is always dominated by the LUT consumption.

In our setting, the amount of resources available for the FPGA-based regex matching engine is 15,000 LUTs and the frequency of the engine is 200 MHz using a 32-bit-wide data path. As explained in [36], the engine containing a single unit (i.e. the single NFA implementation) can achieve the throughput of 6.4 Gbps (200 MHz $\times$ 32 b). Therefore, 16 units are required for the desired link speed of 100 Gbps and 63 units are needed to handle 400 Gbps. With the given amount of LUTs, the size of a single NFA is thus bounded by 937 LUTs (15,000/16) for 100 Gbps and 238 LUTs for 400 Gbps, respectively. These bounds directly limit the complexity of regexes the engine can handle.

We now analyse the resource consumption of the matching engine for two automata, `http-backdoor` ($\mathcal{A}^{\mathrm{RED}}_{\mathtt{bd}}$) and `http-malicious` ($\mathcal{A}^{\mathrm{RED}}_{\mathtt{mal}}$), and evaluate the impact of the reduction techniques. Recall that the automata represent two important sets of know network attacks from Snort [50].

- **100 Gbps**: For this speed, $\mathcal{A}^{\mathrm{RED}}_{\mathtt{mal}}$ can be used without any approximate reduction as it is small enough (it has 382 LUTs) to fit in the available space. On the other hand, $\mathcal{A}^{\mathrm{RED}}_{\mathtt{bd}}$ without the approximate reduction is way too large to fit. (It has 2,266 LUTs and thus at most 6 units fit inside the available space, yielding the throughput of only 38.4 Gbps, which is unacceptable.) The column **LUTs** in Table 3 shows that using our framework, we are able to reduce $\mathcal{A}^{\mathrm{RED}}_{\mathtt{bd}}$ such that it uses 894 LUTs (for $\boldsymbol{k} = 0.3$), and so all of the 16 needed units fit into the FPGA, yielding the throughput over 100 Gbps and the theoretical error bound of a false positive $\leq 3.4 \times 10^{-8}$ wrt the network traffic model $\mathcal{P}_{HTTP}$.
- **400 Gbps**: Regex matching at this speed is extremely challenging. In the case of $\mathcal{A}^{\mathrm{RED}}_{\mathtt{bd}}$, the reduction $\boldsymbol{k} = 0.1$ is required to fit 63 units in the available space. As such a reduction has error bound almost 1, this solution is not useful due to a prohibitively high false positive rate. The situation is better for $\mathcal{A}^{\mathrm{RED}}_{\mathtt{mal}}$. In the exact version, at most 39 units can fit inside the FPGA with the maximum throughput of 249.6 Gbps. On the other hand, when using our reduced automata, we are able to place 63 units into the FPGA, each of the size 224 LUTs ($\boldsymbol{k} = 0.6$), and achieve a throughput of over 400 Gbps with the theoretical error bound of a false positive $\leq 8.7 \times 10^{-8}$ wrt the model $\mathcal{P}_{HTTP}$.

6 Conclusion

We have proposed a novel approach for approximate reduction of NFAs used in network traffic filtering. Our approach is based on a proposal of a probabilistic distance of the original and reduced automaton using a probabilistic model of the input network traffic, which characterizes the significance of particular packets. We characterized the computational complexity of approximate reductions based on the described distance and proposed a sequence of heuristics allowing one to perform the approximate reduction in an efficient way. Our experimental results are quite encouraging and show that we can often achieve a very significant reduction for a negligible loss of precision. We showed that using our approach, FPGA-accelerated network filtering on large traffic speeds can be applied on regexes of malicious traffic where it could not be applied before.

In the future, we plan to investigate other approximate reductions of the NFAs, maybe using some variant of abstraction from abstract regular model checking [7], adapted for the given probabilistic setting. Another important issue for the future is to develop better ways of learning a suitable probabilistic model of the input traffic.

Acknowledgements We thank Jan Kořenek, Vlastimil Košař, and Denis Matoušek for their help with translating regexes into automata and synthesis of FPGA designs, and Martin Žádník for providing us with the backbone network traffic. We thank Stefan Kiefer for helping us proving the **PSPACE** part of Lemma 1 and Petr Peringer for testing our artefact. We also thank the anonymous reviewers for their useful comments, which improved the quality of the paper. The work on this paper was supported by the Czech Science Foundation project 16-24707Y, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the FIT BUT internal project FIT-S-17-4014.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
2. Baier, C., Kiefer, S., Klein, J., Klüppelholz, S., Müller, D., Worrell, J.: Markov chains and unambiguous Büchi automata. In: CAV'16, pp. 23–42. Springer (2016)
3. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: CoNEXT'07, p. 1. ACM (2007)
4. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: ANCS'07, pp. 145–154. ACM (2007)
5. Becchi, M., Wiseman, C., Crowley, P.: Evaluating regular expression matching engines on network and general purpose processors. In: Proceedings of the 5th ACM/IEEE Symposium on Architec-

tures for Networking and Communications Systems, ANCS '09, pp. 30–39. ACM (2009)
6. Benedikt, M., Lenhardt, R., Worrell, J.: Model checking Markov chains against unambiguous Büchi automata. CoRR arXiv:1405.4560v2 (2016)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. STTT **14**(2), 167–191 (2012)
8. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. In: ISCA'06, pp. 191–202. IEEE Computer Society (2006)
9. Bustan, D., Grumberg, O.: Simulation-based minimization. ACM Trans. Comput. Log. **4**(2), 181–206 (2003)
10. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Proceedings of the Second International Colloquium on Grammatical Inference and Applications, ICGI '94, pp. 139–152. Springer (1994)
11. Češka, M., Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Approximate reduction of finite automata for high-speed network intrusion detection. In: Figshare (2018). https://doi.org/10.6084/m9.figshare.5907055
12. Češka, M., Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Approximate reduction of finite automata for high-speed network intrusion detection. In: Proceedings of TACAS'18, *LNCS*, vol. 10806. Springer (2018)
13. Champarnaud, J., Coulon, F.: NFA reduction algorithms by means of regular inequalities. Theor. Comput. Sci. **327**(3), 241–253 (2004)
14. Clark, C.R., Schimmel, D.E.: Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In: FPL'03, Lecture Notes in Computer Science, vol. 2778, pp. 956–959. Springer (2003)
15. Clemente, L.: Büchi automata can have smaller quotients. In: ICALP'11, Lecture Notes in Computer Science, vol. 6756, pp. 258–270. Springer (2011)
16. Csanky, L.: Fast parallel matrix inversion algorithms. In: 16th Annual Symposium on Foundations of Computer Science, pp. 11–12 (1975). https://doi.org/10.1109/SFCS.1975.14
17. Deza, M.M., Deza, E.: Encyclopedia of Distances. Springer, Berlin (2009)
18. Etessami, K.: A hierarchy of polynomial-time computable simulations for automata. In: CONCUR 2002—Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2421, pp. 131–144. Springer (2002)
19. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78, pp. 114–118. ACM, New York, NY, USA (1978). https://doi.org/10.1145/800133.804339
20. Gange, G., Ganty, P., Stuckey, P.J.: Fixing the state budget: approximation of regular languages with small DFAs. In: ATVA'17, Lecture Notes in Computer Science, vol. 10482, pp. 67–83. Springer (2017)
21. Gawrychowski, P., Jez, A.: Hyper-minimisation made efficient. In: MFCS'09, Lecture Notes in Computer Science, vol. 5734, pp. 356–368. Springer (2009)
22. Hartmanns, A., Wendler, P.: TACAS 2018 artifact evaluation VM. In: Figshare (2018). https://doi.org/10.6084/m9.figshare.5896615
23. Hogben, L.: Handbook of Linear Algebra, 2nd edn. CRC Press, Boca Raton (2013)
24. Hopcroft, J.E.: An N log N algorithm for minimizing states in a finite automaton. Technical report (1971)
25. Hutchings, B.L., Franklin, R., Carver, D.: Assisting network intrusion detection with reconfigurable hardware. In: FCCM'02, pp. 111–120. IEEE Computer Society (2002)
26. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM J. Comput. **22**(6), 1117–1141 (1993)
27. Kaštil, J., Kořenek, J., Lengál, O.: Methodology for fast pattern matching by deterministic finite automaton with perfect hashing. In: 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, pp. 823–829 (2009)
28. Kořenek, J., Kobierský, P.: Intrusion detection system intended for multigigabit networks. In: 2007 IEEE Design and Diagnostics of Electronic Circuits and Systems, pp. 1–4 (2007)
29. Kumar, S., Chandrasekaran, B., Turner, J.S., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: ANCS'07, pp. 155–164. ACM (2007)
30. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.S.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: SIGCOMM'06, pp. 339–350. ACM (2006)
31. Kumar, S., Turner, J.S., Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. In: ANCS'06, pp. 81–92. ACM (2006)
32. Liu, C., Wu, J.: Fast deep packet inspection with a dual finite automata. IEEE Trans. Comput. **62**(2), 310–321 (2013)
33. Luchaup, D., De Carli, L., Jha, S., Bach, E.: Deep packet inspection with DFA-trees and parametrized language overapproximation. In: INFOCOM'14, pp. 531–539. IEEE (2014)
34. Malcher, A.: Minimizing finite automata is computationally hard. Theor. Comput. Sci. **327**(3), 375–390 (2004)
35. Maletti, A., Quernheim, D.: Optimal hyper-minimization. CoRR arXiv:1104.3007 (2011)
36. Matoušek, D., Kořenek, J., Puš, V.: High-speed regular expression matching with pipelined automata. In: 2016 International Conference on Field-Programmable Technology (FPT), pp. 93–100 (2016)
37. Mayr, R., Clemente, L.: Advanced automata minimization. In: POPL'13, Transactions on Computer Logic, pp. 63–74. ACM (2013)
38. Mayr, R., et al.: Reduce: A tool for minimizing nondeterministic finite-word and Büchi automata. http://languageinclusion.org/doku.php?id=tools (2017). Accessed 30 Sept 2017
39. Mitra, A., Najjar, W.A., Bhuyan, L.N.: Compiling PCRE to FPGA for accelerating SNORT IDS. In: ANCS'07, pp. 127–136. ACM (2007)
40. Mohri, M.: A disambiguation algorithm for finite automata and functional transducers. In: CIAA'12, pp. 265–277. Springer (2012)
41. Mohri, M.: Edit-distance of weighted automata. In: CIAA'02, Lecture Notes in Computer Science, vol. 2608, pp. 1–23. Springer (2002)
42. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
43. Papadimitriou, C.M.: Computational Complexity. Addison-Wesley, Reading (1994)
44. Parker, A.J., Yancey, K.B., Yancey, M.P.: Regular language distance and entropy. CoRR arXiv:1602.07715 (2016)
45. Puš, V., Tobola, J., Košař, V., Kaštil, J., Kořenek, J.: Netbench: framework for evaluation of packet processing algorithms. In: Symposium On Architecture For Networking And Communications Systems pp. 95–96 (2011)
46. Shützenberger, M.: On the definition of a family of automata. Inf. Control **4**, 245–270 (1961)
47. Sidhu, R.P.S., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: FCCM'01, pp. 227–238. IEEE Computer Society (2001)
48. Solodovnikov, V.I.: Upper bounds on the complexity of solving systems of linear equations. J. Sov. Math. **29**(4), 1482–1501 (1985)
49. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: ISCA'05, pp. 112–122. IEEE Computer Society (2005)
50. The Snort Team: Snort. http://www.snort.org. Accessed 30 Sept 2017

51. Thollard, F., Clark, A.: Learning stochastic deterministic regular languages. In: G. Paliouras, Y. Sakakibara (eds.) Grammatical Inference: Algorithms and Applications: 7th International Colloquium, ICGI 2004, Athens, Greece, October 11–13, 2004. Proceedings, pp. 248–259. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30195-0_22
52. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. In: SFCS '85, pp. 327–338. IEEE
53. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: ANCS'06, pp. 93–102. ACM (2006)

Journal of Automated Reasoning (2021) 65:971–999
https://doi.org/10.1007/s10817-021-09597-w

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

Vojtěch Havlena[1] · Lukáš Holík[1] · Ondřej Lengál[1] · Tomáš Vojnar[1]

Received: 15 April 2020 / Accepted: 20 August 2020 / Published online: 3 August 2021

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

Keywords WS*k*S · Tree automata · Automata term · Finite automata · Monadic second-order logic

1 Introduction

Weak monadic second-order logic of k successors (WS*k*S) is a logic for describing regular properties of finite *k*-ary trees. In addition to talking about trees, WS*k*S can also encode complex properties of a rich class of general graphs by referring to their tree backbones [28]. WS*k*S offers extreme succinctness for the price of non-elementary worst-case complexity. As noticed first by the authors of [16] in the context of WS1S (a restriction that speaks about finite words only), the trade-off between complexity and succinctness may, however, be turned significantly favourable in many practical cases through a use of clever implementation techniques and heuristics. Such techniques were then elaborated in the tool Mona [12,22], the best-known implementation of decision procedures for WS1S and WS2S. Mona has found numerous applications in verification of programs with complex dynamic linked data structures [8,25,26,28,42], string programs [34], array programs [43], parametric systems [3,

Ondřej Lengál
lengal@fit.vutbr.cz

1 IT4I Centre of Excellence, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 00 Brno, Czech Republic

4,6], distributed systems [24,32], hardware verification [2], automated synthesis [18,20,31], and even computational linguistics [29].

Despite the extensive research and engineering effort invested into Mona, due to which it still offers the best all-around performance among existing WS1S/WS2S decision procedures, it is, however, easy to reach its scalability limits. Particularly, Mona implements the classical WS1S/WS2S decision procedures that build a word/tree automaton representing models of the given formula and then check emptiness of the automaton's language. The non-elementary complexity manifests in that the size of the automaton is prone to explode, which is caused mainly by the repeated determinisation (needed to handle negation and alternation of quantifiers) and synchronous product construction (used to handle conjunctions and disjunctions). Users of WSkS are then forced to either find workarounds, such as in [26], or, often restricting the input of their approach, give up using WSkS altogether [38].

As in Mona, we further consider WS2S only (this does not change the expressive power of the logic since k-ary trees can be easily encoded into binary ones). We revisit the use of tree automata (TAs) in the WS2S decision procedure and obtain a new decision procedure that is much more efficient in certain cases. It is inspired by works on *antichain algorithms* for efficient testing of universality and language inclusion of finite automata [1,5,11,39], which implement the operations of testing emptiness of a complement (universality) or emptiness of a product of one automaton with the complement of the other one (language inclusion) via an *on-the-fly* determinisation and product construction. The on-the-fly approach allows one to achieve significant savings by pruning the state space that is irrelevant for the language emptiness test. The pruning is achieved by early termination when detecting non-emptiness (which represents a simple form of *lazy evaluation*), and *subsumption* (which basically allows one to disregard proof obligations that are implied by other ones). Antichain algorithms and their generalizations have shown great efficiency improvements in applications such as abstract regular model checking [5], shape analysis [17], LTL model checking [40], or game solving [41].

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

Our approach is a generalization of our earlier work [13] on WS1S. Although the term structure and the generalized algorithm may seem close to [13], the reasoning behind it is significantly more involved. Particularly, [13] is based on defining the semantics (language) of terms as a function of the semantics of their sub-terms. For instance, the semantics of the term $\{q_1, \ldots, q_n\}$ is defined as the union of languages of the state terms $q_1, \ldots, q_n$, where the language of a state of the base automaton consists of the words *accepted at that state*. With TAs, it is, however, not meaningful to talk about trees accepted from a leaf state, instead, we need to talk about a given state and its *context*, i.e., other states that could be obtained via a bottom-up traversal over the given set of symbols. Indeed, trees have multiple leafs, which may be accepted by a number of different states, and so a tree is *accepted from a set of states*, not from any single one of them alone. We therefore cannot define the semantics of a state term as a tree language, and so we cannot define the semantics of an automata term as the union of the languages of its state sub-terms. This problem seems critical at first because without a sensible notion of the meaning of terms, a straightforward generalization of the algorithm of [13] to trees is not possible. The solution we present here is based on defining the semantics of terms *not* as functions of languages of their sub-terms, but, instead, via the automata constructions they represent.

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

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

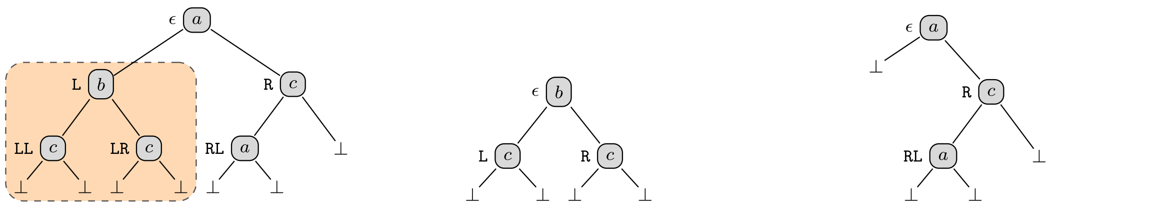
This paper is an extended version of the paper with the same name that appeared in the proceedings of CADE-27 [19], containing more examples and complete proofs of the presented lemmas and theorems, as well as one more optimization of our efficient decision procedure (cf. Sect. 4.5).

2 Preliminaries

In this section, we introduce basic notation, trees, and tree automata, and give a quick introduction to the *weak monadic second-order logic of two successors* (WS2S) and its classical decision procedure. We give the minimal syntax of WS2S only; see, e.g., Comon et al. [9] for more details.

2.1 Basics, Trees, and Tree Automata

Let Σ be a finite set of symbols, called an *alphabet*. The set Σ^* of *words* over Σ consists of finite sequences of symbols from Σ. The *empty word* is denoted by ϵ, with $\epsilon \notin \Sigma$. The *concatenation* of two words u and v is denoted by $u.v$ or simply uv. The *domain* of a partial function $f : X \to Y$ is the set $\mathrm{dom}(f) = \{x \in X \mid \exists y : x \mapsto y \in f\}$, its *image* is the set $\mathrm{img}(f) = \{y \in Y \mid \exists x : x \mapsto y \in f\}$, and its *restriction* to a set Z is the function



(a) A tree τ over Σ **(b)** A tree μ used for the derivative **(c)** A tree τ' from the derivative of τ with respect to $\{\mu\}$

Fig. 1 An example of the derivative. Consider trees τ and μ over the alphabet $\Sigma = \{a, b, c\}$ given in (**a**) and (**b**) respectively. The derivative of τ with respect to $\{\mu\}$ is the set $\{\tau, \tau'\}$ where τ' is given in (**c**)

$f_{|Z} = f \cap (Z \times Y)$. For a binary operator $\bullet$, we write $A\,[\bullet]\,B$ to denote the augmented product $\{a \bullet b \mid (a, b) \in A \times B\}$ of A and B.

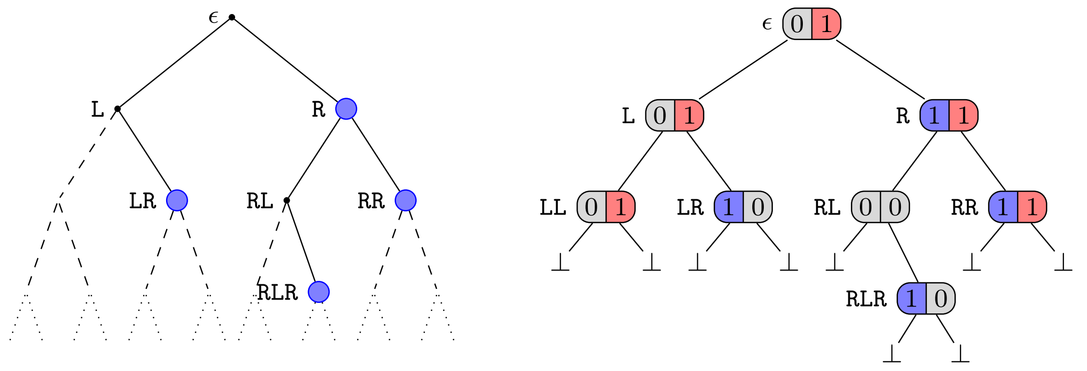
We will consider ordered binary trees. We call a word $p \in \{\mathtt{L}, \mathtt{R}\}^*$ a tree *position* and $p.\mathtt{L}$ and $p.\mathtt{R}$ its *left* and *right child*, respectively. Given an alphabet Σ such that $\perp \notin \Sigma$, a *tree* over Σ is a finite partial function $\tau : \{\mathtt{L},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

The *sub-tree* of τ rooted at a position $p \in \mathrm{dom}(\tau)$ is the tree $\tau' = \{p' \mapsto \tau(p.p') \mid p.p' \in \mathrm{dom}(\tau)\}$. A *prefix* of τ is a tree τ' such that $\tau'_{|\mathrm{dom}(\tau')\setminus \mathit{leaf}(\tau')} \subseteq \tau_{|\mathrm{dom}(\tau)\setminus \mathit{leaf}(\tau)}$. The *derivative* of a tree τ with respect to a set of trees $S \subseteq \Sigma^{木}$ is the set $\tau - S$ of all prefixes τ' of τ such that, for each position $p \in \mathit{leaf}(\tau')$, the sub-tree of τ at p either belongs to S or it is a leaf of τ. Intuitively, $\tau - S$ are all prefixes of τ obtained from τ by removing some of the sub-trees in S. The derivative of a set of trees $T \subseteq \Sigma^{木}$ with respect to S is the set $\bigcup_{\tau \in T}(\tau - S)$. See Fig. 1 for an example of the derivative.

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

2.2 Syntax and Semantics of WS2S

WS2S is a logic that allows quantification over second-order *variables*, which are denoted by upper-case letters $X, Y, \ldots$ and range over *finite sets* of tree positions in $\{\mathtt{L}, \mathtt{R}\}^*$ (the finiteness of variable assignments is reflected in the name *weak*). See Fig. 2a for an example of a set of positions assigned to a variable. Atomic formulae (atoms) of WS2S are of the form: (i) $X \subseteq Y$, (ii) $X = S_\mathtt{L}(Y)$, and (iii) $X = S_\mathtt{R}(Y)$. Informally, the $S_\mathtt{L}(Y)$ function returns all positions from Y shifted to their left child and the $S_\mathtt{R}(Y)$ function returns all positions



(a) Positions assigned to the variable X

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

Fig. 2 An example of an assignment ν to a pair of variables $\{X, Y\}$ such that $\nu(X) = \{\mathtt{LR}, \mathtt{R}, \mathtt{RLR}, \mathtt{RR}\}$ and $\nu(Y) = \{\epsilon, \mathtt{L}, \mathtt{LL}, \mathtt{R}, \mathtt{RR}\}$ and its encoding into a tree

from Y shifted to their right child. Formulae are constructed from atoms using the logical connectives $\wedge$, $\vee$, $\neg$, and the quantifier $\exists\mathbb{X}$ where $\mathbb{X}$ is a finite set of variables (we write $\exists X$ when $\mathbb{X}$ is the singleton set $\{X\}$). Other connectives (such as $\Rightarrow$ or $\forall$) and predicates (such as the predicate $\mathrm{Sing}(X)$ for the singleton set X) can be obtained as syntactic sugar (e.g., we can define the emptiness predicate $X = \emptyset$ as $\forall Y.\, X \subseteq Y$ and the singleton predicate $\mathrm{Sing}(X)$ as $\forall Y.\, Y \subseteq X \Rightarrow (Y = X \vee Y = \emptyset)$).

A *valuation* of a set of variables $\mathbb{X}$ is an assignment $\nu : \mathbb{X} \to 2^{\{\mathtt{L},\mathtt{R}\}^*}$ of $\mathbb{X}$ to finite subsets of $\{\mathtt{L}, \mathtt{R}\}^*$. We use $\nu \triangleleft \{X \mapsto S\}$ to denote a valuation obtained from ν by changing the assignment of X to S. A *model* of a WS2S formula $\varphi(\mathbb{X})$ with the set of free variables $\mathbb{X}$ is a valuation of $\mathbb{X}$ for which the formula is *satisfied*, written $\nu \models \varphi$. Satisfaction of formulae is defined as follows:

(i) $\nu \models X \subseteq Y$ if and only if $\nu(X)$ is a subset of $\nu(Y)$,
(ii) $\nu \models X = S_{\mathtt{L}}(Y)$ if and only if $\nu(X)$ is $\{p.\mathtt{L} \mid p \in \nu(Y)\}$,
(iii) $\nu \models X = S_{\mathtt{R}}(Y)$ if and only if $\nu(X)$ is $\{p.\mathtt{R} \mid p \in \nu(Y)\}$,
(iv) $\nu \models \neg\varphi$ if and only if not $\nu \models \varphi$,
(v) $\nu \models \varphi \wedge \psi$ if and only if $\nu \models \varphi$ and $\nu \models \psi$,
(vi) $\nu \models \varphi \vee \psi$ if and only if $\nu \models \varphi$ or $\nu \models \psi$, and
(vii) $\nu \models \exists X.\, \varphi$ if and only if there is a finite $S \subseteq \{\mathtt{L}, \mathtt{R}\}^*$ such that $\nu \triangleleft \{X \mapsto S\} \models \varphi$.

A formula φ is *valid*, written $\models \varphi$, if and only if all assignments of its free variables are its models, and *satisfiable* if it has a model. Without loss of generality, we assume that each variable in a formula either has only free occurrences or is quantified exactly once; we denote the set of (free and quantified) variables occurring in a formula φ as $\mathit{Vars}(\varphi)$.

2.3 Representing Models as Trees

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

A finite assignment $\nu : \mathbb{X} \to 2^{\{\mathtt{L},\mathtt{R}\}^*}$ of φ's variables can be encoded as a finite tree τ_ν of symbols over $\mathbb{X}$ where every position $p \in \{\mathtt{L}, \mathtt{R}\}^*$ satisfies the following conditions: (a) if $p \in \nu(X)$, then $\tau_\nu(p)$ contains $\{X \mapsto 1\}$, and (b) if $p \notin \nu(X)$, then either $\tau_\nu(p)$

contains $\{X \mapsto 0\}$ or $\tau_\nu(p') = \bot$ for some prefix p' of p (note that the occurrences of $\bot$ in τ are limited since τ still needs to be a tree). Observe that ν can have multiple encodings: the unique minimum one τ_ν^{min} and (infinitely many) extensions of τ_ν^{min} with $\mathbf{0}$-only trees. The *language* of φ is defined as the set of all encodings of its models $\mathcal{L}(\varphi) = \{\tau_\nu \in \Sigma_{\mathbb{X}}^{\divideontimes} \mid \nu \models \varphi \text{ and } \tau_\nu \text{ is an encoding of } \nu\}$. See Fig. 2 for an example of an assignment and its encoding.

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

2.4 The Classical Decision Procedure for WS2S

The classical decision procedure for the WS2S logic goes through a direct construction of a TA $\mathcal{A}_\varphi$ having the same language as a given formula φ. Let us briefly recall the automata constructions used (cf. [9]). Given a complete TA $\mathcal{A} = (Q, \delta, I, R)$, the *complement* assumes that $\mathcal{A}$ is deterministic and returns $\mathcal{A}^{\complement} = (Q, \delta, I, Q \setminus R)$, the projection returns $\pi_X(\mathcal{A}) = (Q, \delta^{\pi_X}, I, R)$ with $\delta_a^{\pi_X}(q, r) = \delta_{\pi_X(a)}(q, r)$, and the *subset construction* returns the deterministic and complete automaton $\mathcal{A}^{\mathcal{D}} = (2^Q, \delta^{\mathcal{D}}, \{I\}, R^{\mathcal{D}})$ where $\delta_a^{\mathcal{D}}(S, S') = \bigcup_{q \in S, q' \in S'} \delta_a(q, q')$ and $R^{\mathcal{D}} = \{S \subseteq Q \mid S \cap R \neq \emptyset\}$. The binary operators $\circ \in \{\cup, \cap\}$ are implemented through a *product construction*, which—given the TA $\mathcal{A}$ and another complete TA $\mathcal{A}' = (Q', \delta', I', R')$—returns the automaton $\mathcal{A} \circ \mathcal{A}' = (Q \times Q', \Delta^{\times}, I^{\times}, R^{\circ})$ where $\Delta_a^{\times}((q, r), (q', r')) = \Delta_a(q, q') \times \Delta_a'(r, r')$, $I^{\times} = I \times I'$, and for $(q, r) \in Q \times Q'$, $(q, r) \in R^{\cap} \Leftrightarrow q \in R \wedge r \in R'$ and $(q, r) \in R^{\cup} \Leftrightarrow q \in R \vee r \in R'$. Testing non-emptiness of $\mathcal{A}$ can be implemented through the equivalence $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if $\mathit{reach}_\delta(I) \cap R \neq \emptyset$ where the set $\mathit{reach}_\delta(S)$ of states *reachable* from a set $S \subseteq Q$ through δ-transitions is computed as the least fixpoint

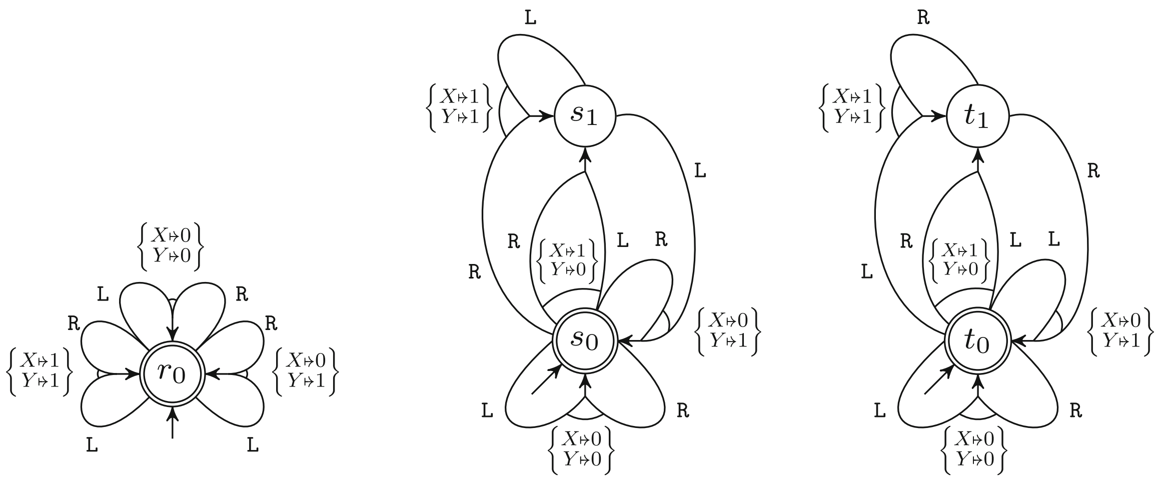
$$\mathit{reach}_\delta(S) = \mu Z.\, S \cup \bigcup_{q,r \in Z} \delta(q, r). \tag{1}$$

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

The classical WSkS decision procedure uses the above operations to construct the automaton $\mathcal{A}_\varphi$ inductively to the structure of φ as follows: (i) If φ is an atomic formula, then $\mathcal{A}_\varphi$ is a pre-defined *base* TA over $\Sigma_{\mathbb{X}}$ (we show those TAs in Fig. 3). (ii) If $\varphi = \varphi_1 \wedge \varphi_2$, then $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cap \mathcal{A}_{\varphi_2}$. (iii) If $\varphi = \varphi_1 \vee \varphi_2$, then $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cup \mathcal{A}_{\varphi_2}$. (iv) If $\varphi = \neg\psi$, then $\mathcal{A}_\varphi = \mathcal{A}_\psi^{\complement}$. (v) Finally, if $\varphi = \exists X.\, \psi$, then $\mathcal{A}_\varphi = (\pi_X(\mathcal{A}_\psi))^{\mathcal{D}} - \mathbf{0}^{\divideontimes}$.

Points (i) to (iv) are self-explanatory. In point (v), the projection implements the quantification by forgetting the values of the X component of all symbols. Since this yields

[1] Note that our definition of projection differs from the usual one, which would create a symbol over the alphabet $\mathbb{X} \setminus \mathbb{Y}$; in the example, it would produce a single symbol $\{Y \mapsto 0\}$ over the alphabet of symbols over $\{Y\}$.


r_0
s_0
s_1
t_0
t_1
L
R
$\{X \mapsto 0, Y \mapsto 0\}$
$\{X \mapsto 1, Y \mapsto 1\}$
$\{X \mapsto 0, Y \mapsto 1\}$
$\{X \mapsto 1, Y \mapsto 0\}$

(a) $\mathcal{A}_{X \subseteq Y}$ **(b)** $\mathcal{A}_{X = S_{\mathtt{L}}(Y)}$ **(c)** $\mathcal{A}_{X = S_{\mathtt{R}}(Y)}$

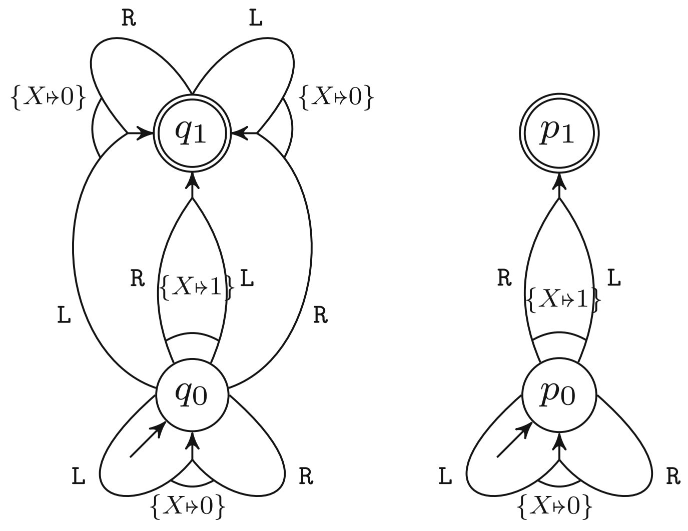
Fig. 3 Tree automata for atomic WS2S formulae. Transitions are represented using multiple-source hyper-edges. For instance, the transition $(s_0, s_1) \xrightarrow{\{X \mapsto 1, Y \mapsto 1\}} s_1$ in $\mathcal{A}_{X = S_{\mathtt{L}}(Y)}$ is represented by the hyper-edge with sources s_0 and s_1 over the symbol $\{X \mapsto 1, Y \mapsto 1\}$ that joins just before entering s_1. The L and R labels on the "legs" of the hyper-edge going to s_0 and s_1 denote the position in the left-hand side of the transition (L and R stand for "left" and "right")

non-determinism, projection is followed by determinisation by the subset construction. Further, the projection can produce some new trees that contain **0**-only labelled sub-trees, which need not be present in some smaller encodings of the same model. Consider, for example, a formula ψ having the language $\mathcal{L}(\psi)$ given by the tree τ_ν in Fig. 2b and all its **0**-extensions. To obtain $\mathcal{L}(\exists X.\psi)$, it is not sufficient to make the projection $\pi_X(\mathcal{L}(\psi))$ because the projected language does not contain the minimum encoding τ_ν^{min} of $\nu : Y \mapsto \{\epsilon, \mathtt{L}, \mathtt{LL}, \mathtt{R}, \mathtt{RR}\}$, but only those encodings ν' such that $\nu'(\mathtt{RLR}) = \{Y \mapsto 0\}$. Therefore, the **0**-derivative is needed to saturate the language with *all* encodings of the encoded models (if some of these encodings were missing, the inductive construction could produce a wrong result, for instance, if the language were subsequently complemented). As mentioned above, on the level of automata, the **0** derivative can be achieved by replacing the set of leaf states I of $\mathcal{A}_\varphi$ by $reach_{\Delta_{\mathbf{0}}}(I)$ where Δ is the transition function of $\mathcal{A}_\varphi$. See [9] for more details.

3 Automata Terms

Our algorithm for deciding WS2S may be seen as an alternative implementation of the classical procedure from Sect. 2.4. The main innovation is the data structure of *automata terms*, which implicitly represent the automata constructed by the automata operations. Unlike the classical procedure—which proceeds by a bottom-up traversal on the formula structure, building an automaton for each sub-formula before proceeding upwards—automata terms allow for constructing parts of automata at higher levels from parts of automata on the lower levels even though the construction of the lower level automata has not yet finished. This allows one to test the language emptiness on the fly and use techniques of state space pruning, which will be discussed later in Sect. 4.

Fig. 4 Tree automata for the predicates used in Example 1



(a) $\mathcal{A}_{\mathrm{Sing}(X)}$ **(b)** $\mathcal{A}_{X=\{\epsilon\}}$

3.1 Syntax of Automata Terms

Terms are created according to the grammar

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

starting from states $q \in Q_i$, denoted as *atomic states*, of a given finite set of *base automata* $\mathcal{B}_i = (Q_i, \delta_i, I_i, R_i)$ with pairwise disjoint sets of states. For simplicity, we assume that the base automata are complete, and we denote by $\mathcal{B} = (Q^{\mathcal{B}}, \delta^{\mathcal{B}}, I^{\mathcal{B}}, R^{\mathcal{B}})$ their component-wise union. *Automata terms A* specify the set of leaf states of an automaton. *Set terms S* list a finite number of the leaf states explicitly, while *derivative terms D* specify them symbolically as states reachable from a set of states S via $\mathbf{0}$'s. The states themselves are represented by *state terms t*. (Notice that set terms S and derivate terms D can be both automata terms and state terms.) Intuitively, the structure of state terms records the automata constructions used to create the top-level automaton from states of the base automata. Non-leaf state terms, the state terms' transition function, and root state terms are then defined inductively from base automata as described below in detail. We will normally use t, u to denote terms of all types (unless the type of the term needs to be emphasized).

Example 1 Consider a formula $\varphi \equiv \neg\exists X.\, \mathrm{Sing}(X) \land X = \{\epsilon\}$ and its corresponding automata term $t_\varphi = \left\{ \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \mathbf{0}^{\ast}} \right\}$ (we will show how t_φ was obtained from φ later). For the sake of presentation, we will consider the base automata given in Fig. 4 for the predicates $\mathrm{Sing}(X)$ and $X = \{\epsilon\}$. The term t_φ above denotes the TA $\left((\pi_X(\mathcal{A}_{\mathrm{Sing}(X)} \cap \mathcal{A}_{X=\{\epsilon\}}))^{\mathcal{D}} - \mathbf{0}^{\ast} \right)^{\complement}$ constructed using the automata operations of intersection, projection, subset construction, derivative, and complement. □

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

(a) Root term states

$$\Delta_a(t+u, t'+u') = \Delta_a(t,t')\,[+]\,\Delta_a(u,u') \quad (8)$$
$$\Delta_a(t \,\&\, u, t' \,\&\, u') = \Delta_a(t,t')\,[\&]\,\Delta_a(u,u') \quad (9)$$
$$\Delta_a(\pi_X(t), \pi_X(t')) = \{\pi_X(u) \mid u \in \Delta_{\pi_X(a)}(t,t')\} \quad (10)$$
$$\Delta_a(\overline{t}, \overline{t'}) = \{\overline{u} \mid u \in \Delta_a(t,t')\} \quad (11)$$
$$\Delta_a(S, S') = \left\{ \bigcup_{t \in S, t' \in S'} \Delta_a(t,t') \right\} \quad (12)$$
$$\Delta_a(q,r) = \delta_a^{\mathcal{B}}(q,r) \quad (13)$$

(b) Transitions among compatible state terms

Fig. 5 Semantics of terms

3.2 Semantics of Terms

We will define the denotation of an automata term t as the automaton $\mathcal{A}_t = (Q, \Delta, I, R)$. For a set automata term $t = S$, we define $I = S$, $Q = reach_\Delta(S)$ (i.e., Q is the set of state terms reachable from the leaf state terms), and Δ and R are defined inductively to the structure of t. Particularly, R contains the terms of Q that satisfy the predicate $\mathcal{R}$ defined in Fig. 5a, and Δ is defined in Fig. 5b, with the addition that whenever the rules in Fig. 5b do not apply, then we let $\Delta_a(t, t') = \{\emptyset\}$. The $\emptyset$ here is used as a universal sink state in order to maintain Δ complete, which is needed for automata terms representing complements to yield the expected language. In Figs. 5a, b, the terms t, t', u, u' are arbitrary terms, S, S' are set terms, and $q, r \in Q^{\mathcal{B}}$.

The transitions of Δ for terms of the type $+$, $\&$, π_X, $\overline{\cdot}$, and S are built from the transition function of their sub-terms analogously to how the automata operations of the product union, product intersection, projection, complement, and subset construction, respectively, build the transition function from the transition functions of their arguments (cf. Sect. 2). The only difference is that the state terms stay *annotated* with the particular operation by which they were made (the annotation of the set state terms are the set brackets). The root states are also defined analogously as in the classical constructions.

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

$$(S - \mathbf{0}^{\ast})^{\mathsf{s}} = reach_{\Delta_{\mathbf{0}}}(S) \tag{14}$$

(with $reach_{\Delta_{\mathbf{0}}}(S)$ defined as the fixpoint (1)), and we complete the definition of Δ and $\mathcal{R}$ in Fig. 5a, b with three new rules to be used with a derivative term D:

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

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

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

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

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

3.3 Properties of Terms

In this section, we establish properties of automata terms that we will use later when establishing the correctness of our decision procedure. An implication of the definitions in the previous section, essential for termination of our algorithm in Sect. 4, is that the automata represented by terms indeed have finitely many states. This is a direct consequence of the following lemma.

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

Proof (idea) First, we define the *depth* of a term t, denoted as $d(t)$, inductively as follows: (i) $d(q) = 1$ for $q \in Q^{\mathcal{B}}$, (ii) $d(t_1 \circ t_2) = 1 + \max(d(t_1), d(t_2))$ for $\circ \in \{\&, +\}$, (iii) $d(\diamond t_1) = 1 + d(t_1)$ for $\diamond \in \{\pi_X, \bar{\cdot}\}$, (iv) $d(S) = 1 + \max_{t \in S}(d(t))$, and (v) $d(S - \Gamma^{\ast}) = 1 + d(S)$.

Then, since the number of reachable states in base automata is finite, for a given n there is a finite number of terms of depth at most n. By induction on the depth of terms, we can show that for a pair of terms t_1 and t_2, it holds that for each $t \in \Delta_a(t_1, t_2)$ we have $d(t) \leq \max(d(t_1), d(t_2))$. Therefore, for an automata term S it holds that $reach_{\Delta}(S)$ is finite. □

Let us denote by $\mathcal{L}(t)$ the language $\mathcal{L}(\mathcal{A}_t)$ of the automaton induced by a term t. In the following, we often use the notions of a term expansion and an expanded term. An *expanded term* is a term that does not contain a derivative term as a subterm. *Term expansion* is then defined recursively as follows: (i)$t^{\mathrm{e}} = t$ if t is expanded and (ii) $t^{\mathrm{e}} = (t[u/u^{\mathrm{s}}])^{\mathrm{e}}$ where u is a derivative term of the form $S - \Gamma^{\ast}$ for an expanded term S. Intuitively, the term expansion saturates derivative subterms in a bottom-up manner. Note that the expansion of any automata term A is a set term, i.e., $A^{\mathrm{e}} = \{t_1, \ldots, t_n\}$.

Lemma 2 *Given an automata term t and its expanded term t^{e}, it holds that*

(i) *t^{e} is of a finite size and*
(ii) $\mathcal{L}\left(t^{\mathrm{e}}\right) = \mathcal{L}(t)$.

Proof *(idea) (i)*: This can be easily seen from the fact that term expansion is performed by a bottom-up traversal on the structure of t while substituting derivative terms with their saturations. From the definition of saturation in (14) and Lemma 1, it follows that each such saturation is finite.

(ii): First, note that saturation preserves language, i.e., it holds that

$$\mathcal{L}\left((S - \mathbf{0}^{\ast})\right) = \mathcal{L}\left((S - \mathbf{0}^{\ast})^{\mathrm{s}}\right). \tag{18}$$

The previous fact follows from the definition of derivative automaton in Sect. 2.4. In particular, given $\mathcal{A}_S = (Q, \Delta, S, R)$, we have that

$$\mathcal{A}_S - \mathbf{0}^{\ast} = (Q, \Delta, reach_{\Delta_{\mathbf{0}}}(S), R), \tag{19}$$

which matches the definition of saturation in (14). The lemma follows from the fact that the expansion substitutes terms for saturated terms with equal languages. □

Lemma 3 below shows that languages of terms can be defined from the languages of their sub-terms if the sub-terms are set terms of derivative terms. The terms on the left-hand sides are implicit representations of the automata operations of the respective language operators on the right-hand sides. The main reason why the lemma cannot be extended to all types of sub-terms and yield an inductive definition of term languages is that it is not meaningful to talk about the bottom-up language of an isolated state term that is neither a set term nor a derivative term (which both are also automata terms). This is also one of the main differences from [13], where every term has its own language, which makes the reasoning and the correctness proofs in the current paper significantly more involved.

Lemma 3 *For automata terms A_1, A_2 and a set term S, the following equalities hold:*

$$\mathcal{L}(\{A_1\}) = \mathcal{L}(A_1) \qquad (a)$$

$$\mathcal{L}(\{A_1 + A_2\}) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \qquad (b)$$

$$\mathcal{L}(\{A_1 \mathbin{\&} A_2\}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \qquad (c)$$

$$\mathcal{L}(\{\overline{A_1}\}) = \overline{\mathcal{L}(A_1)} \qquad (d)$$

$$\mathcal{L}(\{\pi_X(A_1)\}) = \pi_X(\mathcal{L}(A_1)) \qquad (e)$$

$$\mathcal{L}(S - \mathbf{0}^{\ast}) = \mathcal{L}(S) - \mathbf{0}^{\ast} \qquad (f)$$

Proof (*a*): We prove the following more general form of (*a*):

$$\mathcal{L}(\{A_1, \ldots, A_n\}) = \mathcal{L}\left(\bigcup_{1 \le i \le n} A_i^{\mathsf{e}}\right). \qquad (20)$$

(Note that $A_1, \ldots, A_n$ are automata terms—i.e., either set terms or derivative terms—so their expanded terms will be set terms.) Intuitively, in this proof we show that determinisation does not change the language of a term. Let us use $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ to denote the TA represented by the term $\bigcup_{1 \le i \le n} A_i^{\mathsf{e}}$.

($\subseteq$) Let τ be a tree. It holds that $\tau \in \mathcal{L}(\{A_1, \ldots, A_n\})$ if and only if $\tau \in \mathcal{L}\left(\{A_1^{\mathsf{e}}, \ldots, A_n^{\mathsf{e}}\}\right)$, i.e., if there is an accepting run ρ on τ in $\mathcal{A}_{\{A_1^{\mathsf{e}}, \ldots, A_n^{\mathsf{e}}\}}$. Note that ρ maps all leaves of τ to the terms from $\{A_1^{\mathsf{e}}, \ldots, A_n^{\mathsf{e}}\}$, i.e., each leaf of τ is labelled by some A_i^{e}, which is a *set* of terms of a lower level (such a set term can be seen as a *macrostate*—i.e., a set of states—from determinisation of TAs). Moreover, for all non-leaf positions $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$, let $\rho(w) = U$, $\rho(w.\mathrm{L}) = U_{\mathrm{L}}$, and $\rho(w.\mathrm{R}) = U_{\mathrm{R}}$. Then, from (12), we have that if $u \in U$, then there exist $u_{\mathrm{L}} \in U_{\mathrm{L}}$ and $u_{\mathrm{R}} \in U_{\mathrm{R}}$ such that $u \in \Delta_{\tau(w)}(u_{\mathrm{L}}, u_{\mathrm{R}})$. Let us define an auxiliary function $\mu(w, u) = (u_{\mathrm{L}}, u_{\mathrm{R}})$ that we will use later. Since ρ is accepting, there is a term $r \in \rho(\epsilon)$ such that $\mathcal{R}(r)$.

We will now use ρ to construct a run ρ' of $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ on τ. The run ρ' will now map positions to a single term as follows: For the root position, we set $\rho'(\epsilon) = r$. Then, given $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$, the labels of children of w are defined as $\rho'(w.\mathrm{L}) = u_{\mathrm{L}}$ and $\rho'(w.\mathrm{R}) = u_{\mathrm{R}}$ where $(u_{\mathrm{L}}, u_{\mathrm{R}}) = \mu(w, \rho'(w))$. As a consequence, we have that $\forall w \in \mathit{leaf}(\tau) : \rho'(w) \in \bigcup_{1 \le i \le n} A_i^{\mathsf{e}}$. Then, for each $w \in \mathrm{dom}(\tau)$, it holds that $\rho'(w) \in \mathit{reach}_{\Delta}(\bigcup_{1 \le i \le n} A_i^{\mathsf{e}})$ where Δ is the transition function of $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$. Therefore, ρ' is a run of $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ on τ and is accepting, so $\tau \in \mathcal{L}\left(\bigcup_{1 \le i \le n} A_i^{\mathsf{e}}\right)$.

($\supseteq$) Consider a tree $\tau \in \mathcal{L}\left(\bigcup_{1 \le i \le n} A_i^{\mathsf{e}}\right)$. Then there is an accepting run ρ on τ in $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$. We can then use ρ to construct the run ρ' on $\mathrm{dom}(\tau)$ defined as follows: For $u \in \mathit{leaf}(\tau)$, if $\rho(u) \in A_i^{\mathsf{e}}$, we set $\rho'(u) = A_i^{\mathsf{e}}$. For $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$, we set $\rho'(w) = r$ such

that $\{r\} = \Delta_{\tau(w)}(\rho'(w.\mathrm{L}), \rho'(w.\mathrm{R}))$ (we know that $\Delta_{\tau(w)}(\rho'(w.\mathrm{L}), \rho'(w.\mathrm{R}))$ is a singleton set due to (12)). For the constructed run ρ', it now holds that $\forall w \in \mathrm{dom}(\tau) : \rho(w) \in \rho'(w)$, therefore ρ' is an accepting run on τ in $\mathcal{A}_{\{A_1^{\mathrm{e}},\ldots,A_n^{\mathrm{e}}\}}$, i.e., $\tau \in \mathcal{L}(\{A_1, \ldots, A_n\})$.
(*b*): ($\subseteq$) Let $\tau \in \mathcal{L}(\{A_1 + A_2\})$. Then there is an accepting run ρ on τ in $\mathcal{A}_{\{A_1^{\mathrm{e}}+A_2^{\mathrm{e}}\}}$. Since ρ is accepting, we can define mappings ρ_1, ρ_2 on $\mathrm{dom}(\tau)$ such that for all $w \in \mathrm{dom}(\tau)$ we have $\rho_1(w) = l(\rho(w))$ and $\rho_2(w) = r(\rho(w))$ where $l(S_1 + S_2) = S_1$ and $r(S_1 + S_2) = S_2$. The mappings ρ_1 and ρ_2 are runs of $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$ and $\mathcal{A}_{\{A_2^{\mathrm{e}}\}}$ on τ respectively. Moreover, since $\mathcal{R}(\rho(\epsilon))$, we have that $\mathcal{R}(\rho_1(\epsilon)) \vee \mathcal{R}(\rho_2(\epsilon))$. To conclude, $\tau \in \mathcal{L}\left(\mathcal{A}_{\{A_1^{\mathrm{e}}\}}\right)$ or $\tau \in \mathcal{L}\left(\mathcal{A}_{\{A_2^{\mathrm{e}}\}}\right)$, so $\tau \in \mathcal{L}(\{A_1\}) \cup \mathcal{L}(\{A_2\})$ and from (*a*) we get $\tau \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

($\supseteq$) Consider $\tau \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. From (*a*) we get $\tau \in \mathcal{L}(\{A_1\}) \cup \mathcal{L}(\{A_2\})$. Then there are runs ρ_1 in $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$ and ρ_2 in $\mathcal{A}_{\{A_2^{\mathrm{e}}\}}$ on τ such that at least one of them is accepting. We can define a mapping ρ on $\mathrm{dom}(\tau)$ such that $\forall w \in \mathrm{dom}(\tau) : \rho(w) = \rho_1(w) + \rho_2(w)$, which is an accepting run on τ in $\mathcal{A}_{\{A_1^{\mathrm{e}}+A_2^{\mathrm{e}}\}}$. Therefore $\tau \in \mathcal{L}(\{A_1 + A_2\})$.
(*c*): Dual to (*b*).
(*d*): Let τ be a tree. We will consider runs ρ and $\overline{\rho}$ of $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$ and $\mathcal{A}_{\overline{\{A_1^{\mathrm{e}}\}}}$ on τ respectively. First, note that both runs exist, which is guaranteed by the presence of the universal sink state $\emptyset$, cf. Sect. 3.2. Second, note that the two runs are unique, since there is a single leaf state and the transition function is deterministic by (12). Further, from (11), it holds that $\forall w \in \mathrm{dom}(\tau) : \overline{\rho}(w) = \overline{\rho(w)}$. From the definition of $\mathcal{R}$ we have $\mathcal{R}(\overline{\rho}(\epsilon)) \Leftrightarrow \neg\mathcal{R}(\rho(\epsilon))$, therefore, ρ is not accepting in $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$ if and only if $\overline{\rho}$ is accepting in $\mathcal{A}_{\overline{\{A_1^{\mathrm{e}}\}}}$. As a consequence, $\tau \in L(\{\overline{A_1^{\mathrm{e}}}\})$ if and only if $\tau \notin L(\{A_1^{\mathrm{e}}\})$. From (*a*), we know that $L(\{A_1^{\mathrm{e}}\}) = L(A_1^{\mathrm{e}})$.
(*e*): ($\subseteq$) Let $\tau \in \mathcal{L}(\{\pi_X(A_1)\})$ and ρ be an accepting run of $\mathcal{A}_{\{\pi_X(A_1^{\mathrm{e}})\}}$ on τ. From the definition of the transition function in (10) and (4), we get that there is an accepting run ρ' on some τ' in $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$ where $\tau \in \pi_X(\tau')$ and $\forall w \in \mathrm{dom}(\tau) : \rho(w) = \pi_X(\rho'(w))$. Therefore, $\tau \in \pi_X(\mathcal{L}(\{A_1\})) = \pi_X(\mathcal{L}(A_1))$.

($\supseteq$) Let $\tau \in \pi_X(\mathcal{L}(A_1))$. From the definition of projection, there is $\tau' \in \mathcal{L}(A_1)$ such that $\tau \in \pi_X(\tau')$. According to (*a*), there is an accepting run ρ on τ' in $\mathcal{A}_{\{A_1^{\mathrm{e}}\}}$. Then there is also an accepting run ρ' on τ in $\mathcal{A}_{\{\pi_X(A_1^{\mathrm{e}})\}}$ where $\forall w \in \mathrm{dom}(\tau) : \rho'(w) = \pi_X(\rho(w))$.
(*f*): We prove the following more general equality: $\mathcal{L}(S) - \Gamma^{\star} = \mathcal{L}\left(S - \Gamma^{\star}\right)$, for a set of symbols Γ (note that S is a set term). In the following text, given a set term U, we define $U \ominus \Gamma = U^{\mathrm{e}} \cup \bigcup\{\Delta_\Gamma(t_1, t_2) \mid t_1, t_2 \in U^{\mathrm{e}}\}$. Note that $\mathit{reach}_\Delta(U^{\mathrm{e}}) = \mathit{reach}_\Delta(U \ominus \Gamma)$. Further, we use $\Gamma^{\leq n}$ to denote the set of trees over Γ of height at most n, i.e., $\Gamma^{\leq n} = \{t \in \Gamma^{\star} \mid \forall w \in \mathrm{dom}(t) : |w| \leq n\}$. We first prove the following two claims.

Claim 1 *Let U be a set term. Then $\mathcal{L}(U \ominus \Gamma) = \mathcal{L}(U) - \Gamma^{\leq 1}$.*

Proof ($\subseteq$) Let $\tau \in \mathcal{L}(U \ominus \Gamma)$ and ρ be an accepting run of $\mathcal{A}_{U\ominus\Gamma}$ on τ. The run ρ maps leaves of τ to the leaf states in $U \ominus \Gamma$. Moreover, for each $w \in \mathit{leaf}(\tau)$ such that $\rho(w) \notin U^{\mathrm{e}}$ (i.e., ρ maps w to a newly added leaf state) there exist $t_{\mathrm{L}}^w, t_{\mathrm{R}}^w \in U^{\mathrm{e}}$ such that $\rho(w) \in \Delta_\Gamma(t_{\mathrm{L}}^w, t_{\mathrm{R}}^w)$. We can therefore extend ρ to the run ρ' defined such that $\rho'_{|\mathrm{dom}(\tau)} = \rho$ and for all $w \in \mathit{leaf}(\tau)$ such that $\rho(w) \notin U^{\mathrm{e}}$, we define $\rho'(w.\mathrm{L}) = t_{\mathrm{L}}^w$ and $\rho'(w.\mathrm{R}) = t_{\mathrm{R}}^w$. The run ρ' is accepting in $\mathcal{A}_{U^{\mathrm{e}}}$ on a tree $\tau' \in \mathcal{L}(U)$ such that $\tau \in \tau' - \Gamma^{\leq 1}$, and so $\tau \in \mathcal{L}(U) - \Gamma^{\leq 1}$.

($\supseteq$) Let $\tau \in \mathcal{L}(U) - \Gamma^{\leq 1}$ and $\tau' \in \mathcal{L}(U)$ be a tree such that $\tau \in \tau' - \Gamma^{\leq 1}$. Hence there is an accepting run ρ' of $\mathcal{A}_{U^{\mathrm{e}}}$ on τ'. Consider the set $\Theta = \{w \in \mathit{leaf}(\tau) \mid \rho'(w) \notin U^{\mathrm{e}}\}$ of positions mapped by ρ' to newly added states. Since $\tau \in \tau' - \Gamma^{\leq 1}$, it holds that $\forall w \in \Theta$: $\rho'(w.\mathrm{L}) \in U^{\mathrm{e}} \wedge \rho'(w.\mathrm{R}) \in U^{\mathrm{e}} \wedge \tau'(w) \in \Gamma$. Therefore, $\rho = \rho'_{|\mathrm{dom}(\tau)}$ is an accepting run of $\mathcal{A}_{U\ominus\Gamma}$ on τ, i.e., $\tau \in \mathcal{L}(U \ominus \Gamma)$. ■

Claim 2 *Let U be a set term, $U_0 = U$, and $U_{i+1} = U_i \ominus \Gamma$ for $i \geq 0$. Then $\mathcal{L}(U_m) = \mathcal{L}(U) - \Gamma^{\leq m}$.*

Proof We prove the claim by induction on m.

- *Base case* $m = 0$: $\mathcal{L}(U_0) = \mathcal{L}(U) = \mathcal{L}(U) - \Gamma^{\leq 0}$.
- *Inductive case*: We assume that the claim holds for $0, \ldots, m$. We prove that it holds also for $m + 1$. From Claim 1 we have

$$\mathcal{L}(U_{m+1}) = \mathcal{L}(U_m \ominus \Gamma) = \mathcal{L}(U_m) - \Gamma^{\leq 1}. \tag{21}$$

By the induction hypothesis we further have

$$\mathcal{L}(U_{m+1}) = (\mathcal{L}(U) - \Gamma^{\leq m}) - \Gamma^{\leq 1}. \tag{22}$$

Finally, from the definition of the derivative we obtain

$$(\mathcal{L}(U) - \Gamma^{\leq m}) - \Gamma^{\leq 1} = \mathcal{L}(U) - \Gamma^{\leq m+1}, \tag{23}$$

which concludes the proof. ■

We now prove the main part of the lemma. Consider the sequence of automata terms $S_0, S_1, \ldots$ where $S_0 = S^{\mathrm{e}}$ and $S_{i+1} = S_i \ominus \Gamma$. From the monotonicity of $\ominus$ and Lemma 1, there is an n such that $S_n \neq S_{n-1}$ and $S_n = S_{n+i}$ for all $i \geq 0$. From Claim 2 we have $\mathcal{L}(S_i) = \mathcal{L}(S) - \Gamma^{\leq i}$ and, consequently, $\mathcal{L}(S_n) = \mathcal{L}(S) - \Gamma^{\leq n}$. Because S_n is the fixpoint of the sequence of automata terms $S_0, S_1, \ldots$, it holds that $\mathcal{L}(S_n) = \mathcal{L}(S) - \Gamma^{*}$. Finally, we have $S_n = reach_{\Delta_\Gamma}(S^{\mathrm{e}}) = S - \Gamma^{*}$ (by (14)), so we conclude that $\mathcal{L}(S) - \Gamma^{*} = \mathcal{L}\left(S - \Gamma^{*}\right)$. □

Lemma 3 shows fundamental properties of terms. Based on it we further focus on flattening of terms, whose properties are described by the following lemma.

Lemma 4 *For sets of terms S and S' such that $S \neq \emptyset$ and $S' \neq \emptyset$, we have:*

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

Proof (a): $(\subseteq)$ Let $\tau \in \mathcal{L}\left(\{S + S'\}\right)$. From Lemma 3$b$ we have $\mathcal{L}\left(\{S + S'\}\right) = \mathcal{L}(S) \cup \mathcal{L}\left(S'\right)$. Hence there are runs ρ_1 in $\mathcal{A}_{S^{\mathrm{e}}}$ and ρ_2 in $\mathcal{A}_{S'^{\mathrm{e}}}$ on τ and, moreover, at least one of them is accepting (both runs exist since the transition function Δ is complete). Then, we can construct a mapping ρ from τ defined such that for all $w \in \mathrm{dom}(\tau)$, we set $\rho(w) = \rho_1(w) + \rho_2(w)$. Note that ρ is a run of $\mathcal{A}_{\{t_1^{\mathrm{e}} + t_2^{\mathrm{e}} \mid t_1 \in S, t_2 \in S'\}}$ on τ, i.e., it maps leaves of $\mathrm{dom}(\tau)$ to terms of the form $t_1^{\mathrm{e}} + t_2^{\mathrm{e}}$ for $t_1 \in S$ and $t_2 \in S'$. Moreover, ρ is accepting since at least one of the runs ρ_1 and ρ_2 is accepting. Therefore, $\tau \in \mathcal{L}\left(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\}\right)$. From the definition of the augmented product, it follows that $\tau \in \mathcal{L}\left(S\,[+]\,S'\right)$ and, finally, from Lemma 3a, we have $\tau \in \mathcal{L}\left(\{S\,[+]\,S'\}\right)$.

$(\supseteq)$ Let $\tau \in \mathcal{L}\left(\{S\,[+]\,S'\}\right)$. From Lemma 3$a$, we get $\tau \in \mathcal{L}\left(S\,[+]\,S'\right)$, and from the definition of the augmented product, we obtain that $\tau \in \mathcal{L}\left(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\}\right)$. Therefore, there is an accepting run ρ on τ in $\mathcal{A}_{\{t_1^{\mathrm{e}} + t_2^{\mathrm{e}} \mid t_1 \in S, t_2 \in S'\}}$. Furthermore, let us consider the run ρ' of $\mathcal{A}_{\{S + S'\}}$ on τ (note that, due to (12) and the completeness of the transition function, there is exactly one). By induction on the structure of τ, we can easily show that for all $w \in \mathrm{dom}(\tau)$, if $\rho(w) = t_1 + t_2$, then $\rho'(w) = S_1 + S_2$ such that $t_1 \in S_1$ and $t_2 \in S_2$ (the property clearly holds at leaves and is also preserved by the transition function). Let $\rho(\epsilon) = t_1^{\epsilon} + t_2^{\epsilon}$ and

$\rho'(\epsilon) = S_1^\epsilon + S_2^\epsilon$. Since $\mathcal{R}(t_1^\epsilon + t_2^\epsilon)$, it also holds that $\mathcal{R}(S_1^\epsilon + S_2^\epsilon)$. Therefore, ρ' is accepting, so $\tau \in \mathcal{L}\left(\{S + S'\}\right)$.

(b): Dual to (a).

(c): From Lemma 3e we have that $\mathcal{L}(\{\pi_X(S)\}) = \pi_X(\mathcal{L}(S))$. Therefore, it is sufficient to prove the following identity: $\pi_X(\mathcal{L}(S)) = \mathcal{L}(\{\pi_X(t) \mid t \in S\})$.

($\subseteq$) Let $\tau \in \pi_X(\mathcal{L}(S))$. Then, there is a tree $\tau' \in \mathcal{L}(S)$ such that $\tau \in \pi_X(\tau')$. Let ρ be an accepting run of $\mathcal{A}_{S^e}$ on τ'. We will construct a run ρ' of $\mathcal{A}_{\{\pi_X(t) \mid t \in S^e\}}$ on τ' such that for all $w \in \mathrm{dom}(\tau)$, we set $\rho'(w) = \pi_X(\rho(w))$. It follows that $\tau \in \mathcal{L}(\{\pi_X(t) \mid t \in S\})$.

($\supseteq$) Let $\tau \in \mathcal{L}(\{\pi_X(t) \mid t \in S\})$ and ρ be an accepting run of $\mathcal{A}_{\{\pi_X(t) \mid t \in S^e\}}$ on τ. We will now construct a mapping ρ' from $\mathrm{dom}(\tau)$ such that for all $w \in \mathrm{dom}(\tau)$, we set $\rho'(w) = t$ where $\rho(w) = \pi_X(t)$. It follows that ρ' is an accepting run of $\mathcal{A}_{S^e}$ on τ', and so $\tau \in \pi_X(\mathcal{L}(S))$. □

3.4 Terms of Formulae

Our algorithm in Sect. 4 will translate a WS2S formula φ into the automata term $t_\varphi = \{\langle\varphi\rangle\}$ representing a deterministic automaton with its only leaf state represented by the state term $\langle\varphi\rangle$. The base automata of t_φ include the automaton $\mathcal{A}_{\varphi_0}$ for each atomic predicate φ_0 used in φ. The state term $\langle\varphi\rangle$ is then defined inductively to the structure of φ as follows:

$$\langle\varphi_0\rangle = I_{\varphi_0} \tag{24}$$
$$\langle\varphi \wedge \psi\rangle = \langle\varphi\rangle \mathbin{\&} \langle\psi\rangle \tag{25}$$
$$\langle\varphi \vee \psi\rangle = \langle\varphi\rangle + \langle\psi\rangle \tag{26}$$
$$\langle\neg\varphi\rangle = \overline{\langle\varphi\rangle} \tag{27}$$
$$\langle\exists X.\, \varphi\rangle = \{\pi_X(\langle\varphi\rangle)\} - \mathbf{0}^{*} \tag{28}$$

In the definition, φ_0 is an atomic predicate, I_{φ_0} is the set of leaf states of $\mathcal{A}_{\varphi_0}$, and φ and ψ denote arbitrary WS2S formulae. We note that the translation rules may create sub-terms of the form $\{\{t\}\}$, i.e., with nested set brackets. Since $\{\cdot\}$ semantically means determinisation by subset construction, such double determinisation terms can be always simplified to $\{t\}$ (cf. Lemma 3a). See Example 1 for a formula φ and its corresponding term t_φ. Theorem 1 establishes the correctness of the formula-to-term translation.

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

Proof To simplify the proof, we restrict the definition of terms to *deterministic terms U* constructed using the following grammar:

$$U ::= \{u, \ldots, u\} \mid \{\pi_X(u), \ldots, \pi_X(u)\} \tag{29}$$

$$u ::= q \mid u + u \mid u \mathbin{\&} u \mid \overline{u} \mid U \mid U - \Gamma^{*} \tag{30}$$

where q is a state of an automaton. It is easy to see that deterministic terms form a proper subset of all terms constructed using the definition in Sect. 3.1 (e.g., the term $\pi_X(t_1) \mathbin{\&} \pi_X(t_2)$ is not deterministic). They are, however, sufficient to capture the terms that emerge from the translation presented above. Note that for two expanded deterministic terms t_1 and t_2 we have $|\Delta_a(t_1, t_2)| = 1$. Further note that for a WS2S formula φ, $\langle\varphi\rangle$ is a deterministic term.

Now, we prove $\mathcal{L}(\varphi) = \mathcal{L}(\{\langle\varphi\rangle\})$ by induction on the structure of φ. In the proof, we use properties of the classical decision procedure from Sect. 2.4.

– $\varphi = \varphi_0$ where φ_0 is an atomic formula: Let I_{φ_0} be the set of leaf states of $\mathcal{A}_{\varphi_0}$.

$$\begin{aligned}
\mathcal{L}(\{\langle\varphi_0\rangle\}) &= \mathcal{L}\left(\{I_{\varphi_0}\}\right) && \wr(24)\wr \\
&= \mathcal{L}\left(I_{\varphi_0}\right) && \wr\text{Lemma } 3a\wr \\
&= \mathcal{L}\left(\mathcal{A}_{\varphi_0}\right) && \wr\text{term semantics}\wr \\
&= \mathcal{L}(\varphi_0)\,. && \wr\text{property of automata for atoms}\wr
\end{aligned}$$

– $\varphi = \psi_1 \wedge \psi_2$: We use the following equational reasoning:

$$\begin{aligned}
\mathcal{L}(\{\langle\psi_1 \wedge \psi_2\rangle\}) &= \mathcal{L}(\{\langle\psi_1\rangle \,\&\, \langle\psi_2\rangle\}) && \wr(25)\wr \\
&= \mathcal{L}(\{\{\langle\psi_1\rangle \,\&\, \langle\psi_2\rangle\}\}) && \wr\text{Lemma } 3a\wr \\
&= \mathcal{L}(\{\{\langle\psi_1\rangle\} \,\&\, \{\langle\psi_2\rangle\}\}) && \wr\text{Lemma } 4b\wr \\
&= \mathcal{L}(\{\langle\psi_1\rangle\}) \cap \mathcal{L}(\{\langle\psi_2\rangle\})\,. && \wr\text{Lemma } 3c\wr \\
&= \mathcal{L}(\psi_1) \cap \mathcal{L}(\psi_2) && \wr\text{induction hypothesis}\wr \\
&= \mathcal{L}(\varphi)\,. && \wr\text{classical procedure}\wr
\end{aligned}$$

– $\varphi = \psi_1 \vee \psi_2$:
We use the following equational reasoning:

$$\begin{aligned}
\mathcal{L}(\{\langle\psi_1 \vee \psi_2\rangle\}) &= \mathcal{L}(\{\langle\psi_1\rangle + \langle\psi_2\rangle\}) && \wr(26)\wr \\
&= \mathcal{L}(\{\{\langle\psi_1\rangle + \langle\psi_2\rangle\}\}) && \wr\text{Lemma } 3a\wr \\
&= \mathcal{L}(\{\{\langle\psi_1\rangle\} + \{\langle\psi_2\rangle\}\}) && \wr\text{Lemma } 4a\wr \\
&= \mathcal{L}(\{\langle\psi_1\rangle\}) \cup \mathcal{L}(\{\langle\psi_2\rangle\})\,. && \wr\text{Lemma } 3b\wr \\
&= \mathcal{L}(\psi_1) \cup \mathcal{L}(\psi_2) && \wr\text{induction hypothesis}\wr \\
&= \mathcal{L}(\varphi)\,. && \wr\text{classical procedure}\wr
\end{aligned}$$

– $\varphi = \neg\psi$: First, we prove the following claim:

Claim 3 *Let t be a deterministic term, then* $\mathcal{L}\left(\{\overline{\{t\}}\}\right) = \mathcal{L}\left(\{\bar{t}\}\right)$.

Proof First, consider two expanded deterministic terms t_1 and t_2. Since t_1 and t_2 are deterministic, from (12) we have $\Delta_a(t_1, t_2) = \{t'\}$ for some deterministic term t' and any symbol a. Therefore (from (11)), $\Delta_a(\overline{t_1}, \overline{t_2}) = \{\overline{t'}\}$ and $\Delta_a(\overline{\{t_1\}}, \overline{\{t_2\}}) = \{\overline{\{t'\}}\}$. Hence, there is an accepting run ρ on a tree τ in $\mathcal{A}_{\{\overline{\{t\}}\}}$ if and only if there is an accepting run ρ' on τ in $\mathcal{A}_{\{\bar{t}\}}$ where for all $w \in \mathrm{dom}(\tau)$ it holds that $\rho(w) = \bar{s} \Leftrightarrow \rho'(w) = \overline{\{s\}}$. ■

We proceed to the main part of the proof.

$$\begin{aligned}
\mathcal{L}(\{\langle\neg\psi\rangle\}) &= \mathcal{L}\left(\{\overline{\langle\psi\rangle}\}\right) && \wr(27)\wr \\
&= \mathcal{L}\left(\{\overline{\{\langle\psi\rangle\}}\}\right) && \wr\text{Claim } 3\wr \\
&= \overline{\mathcal{L}(\{\langle\psi\rangle\})} && \wr\text{Lemma } 3d\wr \\
&= \overline{\mathcal{L}(\psi)} && \wr\text{induction hypothesis}\wr \\
&= \mathcal{L}(\varphi)\,. && \wr\text{classical procedure}\wr
\end{aligned}$$

– $\varphi = \exists X.\, \psi$: We start by proving the following claim:

Claim 4 *Let t be a deterministic term, then* $\mathcal{L}(\{\pi_X(\{t\})\}) = \mathcal{L}(\{\pi_X(t)\})$.

Proof First, consider two expanded deterministic terms t_1 and t_2. Since t_1 and t_2 are both deterministic, we have $\Delta_a(t_1, t_2) = \{t_a\}$ for some deterministic term t_a and any symbol a. Therefore, according to (10), $\Delta_a(\pi_X(t_1), \pi_X(t_2)) = \{\pi_X(t_b) \mid b \in \pi_X(a)\}$ and $\Delta_a(\pi_X(\{t_1\}), \pi_X(\{t_2\})) = \{\pi_X(\{t_b\}) \mid b \in \pi_X(a)\}$. Hence, there is an accepting run ρ on a tree τ in $\mathcal{A}_{\{\pi_X(\{t\})\}}$ if and only if there is an accepting run ρ' on τ in $\mathcal{A}_{\{\pi_X(t)\}}$, where for all $w \in \mathrm{dom}(\tau)$ it holds that $\rho(w) = \pi_X(s) \Leftrightarrow \rho'(w) = \pi_X(\{s\})$. ■

We proceed to the main part of the proof.

$$\begin{aligned}
\mathcal{L}(\{\langle \exists X.\, \psi\rangle\}) &= \mathcal{L}\left(\{\pi_X(\langle\psi\rangle)\} - \mathbf{0}^{*}\right) && \wr(28)\int \\
&= \mathcal{L}(\{\pi_X(\langle\psi\rangle)\}) - \mathbf{0}^{*} && \wr\text{Lemma } 3f\int \\
&= \mathcal{L}(\{\pi_X(\{\langle\psi\rangle\})\}) - \mathbf{0}^{*} && \wr\text{Claim } 4\int \\
&= \pi_X(\mathcal{L}(\{\langle\psi\rangle\})) - \mathbf{0}^{*} && \wr\text{Lemma } 3e\int \\
&= \pi_X(\mathcal{L}(\psi)) - \mathbf{0}^{*} && \wr\text{induction hypothesis}\int \\
&= \mathcal{L}(\varphi)\,. && \wr\text{classical procedure}\int
\end{aligned}$$

□

4 An Efficient Decision Procedure

The development in Sect. 3 already implies a naive automata term-based satisfiability check. Namely, by Theorem 1, we know that a formula φ is satisfiable if and only if $\mathcal{L}(\mathcal{A}_{t_\varphi}) \neq \emptyset$. After translating φ into t_φ using rules (24)–(28), we may use the definitions of the transition function and root states of $\mathcal{A}_{t_\varphi} = (Q, \Delta, I, F)$ in Sect. 3 to decide the language emptiness through evaluating the root state test $\mathcal{R}(reach_\Delta(I))$. The equalities and equivalences (8)–(17) can be implemented as recursive functions. We will further refer to this algorithm as the *simple recursion*. The evaluation of $reach_\Delta(I)$ induces nested evaluations of the fixpoint (14): the one on the top level of the language emptiness test and another one for every expansion of a derivative sub-term. The termination of these fixpoint computations is guaranteed due to Lemma 1.

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

4.1 Memoization

The simple recursion repeats the fixpoint computations that saturate derivative terms from scratch at every call of the transition function or root test. This is easily countered through *memoization*, known, e.g., from compilers of functional languages, which caches results of function calls in order to avoid their re-evaluation. Namely, after saturating a derivative sub-term $t = S - \mathbf{0}^{\ast}$ of t_φ for the first time, we simply *replace* t in t_φ by the saturation $t^{\mathrm{s}} = reach_{\Delta_{\mathbf{0}}}(S)$. Since a derivative is a symbolic representation of its saturated version (cf. (14)), the replacement does not change the language of t_φ. Using memoization, every fixpoint computation is then carried out only once.

4.2 Lazy Evaluation

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

Furthermore, *early termination* is used to optimize fixpoint computations used to saturate derivatives within tests $\mathcal{R}(S - \mathbf{0}^{\ast})$ (obtained from sub-formulae such as $\exists X.\ \psi$). Namely, instead of first unfolding the whole fixpoint into a set $\{t_1, \ldots, t_n\}$ and only then testing whether $\mathcal{R}(t_i)$ is true for some t_i, the terms t_i can be tested as soon as they are computed, and the fixpoint computation can be stopped early, immediately when the test succeeds on one of them. Then, instead of replacing the derivative sub-term by its full saturation, we replace it by the partial result $\{t_1, \ldots, t_i\} - \mathbf{0}^{\ast}$ for $i \le n$. Finishing the evaluation of the fixpoint computation might later be required in order to compute a transition from the derivative. We note that this corresponds to the concept of *continuations* from functional programming, used to represent a paused computation that may be required to continue later.

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

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

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

4.3 Subsumption

The next technique we use is based on pruning out parts of a search space that are *subsumed* by other parts. In particular, we generalize (in a similar way as we did for WS1S in our previous work [13]) the concept used in *antichain* algorithms for efficiently deciding language inclusion and universality of finite word and tree automata [1,5,11,39]. Although the problems are in general computationally infeasible (they are PSPACE-complete for finite word automata and EXPTIME-complete for finite tree automata), antichain algorithms can solve them efficiently in many practical cases.

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

$$\{t_1, t_2, \ldots, t_n\} \rightsquigarrow \{t_2, \ldots, t_n\} \qquad \text{for } t_1 \sqsubseteq t_2, \tag{31}$$

which may be used to simplify set sub-terms of automata terms. The rule (31) is applied after every iteration of the fixpoint computation on the current partial result. Hence the sequence of partial results is monotone, which, together with the finiteness of $reach_\Delta(t)$, guarantees termination. The *subsumption* relation $\sqsubseteq$ used in the rule is defined as

$$S \sqsubseteq S' \Leftrightarrow S \subseteq S' \vee S \sqsubseteq^{\forall\exists} S' \tag{32}$$

$$t \,\&\, u \sqsubseteq t' \,\&\, u' \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{33}$$

$$t + u \sqsubseteq t' + u' \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{34}$$

$$\overline{t} \sqsubseteq \overline{t'} \Leftrightarrow t' \sqsubseteq t \tag{35}$$

$$\pi_X(t) \sqsubseteq \pi_X(t') \Leftrightarrow t \sqsubseteq t' \tag{36}$$

where $S \sqsubseteq^{\forall\exists} S'$ denotes $\forall t \in S\ \exists t' \in S'. t \sqsubseteq t'$. Intuitively, on base TAs, subsumption corresponds to inclusion of the set terms (the left disjunct of (32)). This clearly has the intended outcome: a larger set of states can always simulate a smaller set in accepting a tree. The rest of the definition is an inductive extension of the base case. It can be shown that $\sqsubseteq$ for any automata term t is an upward simulation on $\mathcal{A}_t$ in the sense of [1]. Consequently, rewriting sub-terms in an automata term according to the new rule (31) does not change its language.

4.4 Product Flattening

Product flattening is a technique that we use to reduce the size of fixpoint saturations that generate conjunctions and disjunctions of sets as their elements. Consider a term of the form $D = \{\pi_X(S_0 \,\&\, S'_0)\} - \mathbf{0}^{\bigstar}$ for a pair of sets of terms S_0 and S'_0 where the TAs $\mathcal{A}_{S_0}$ and $\mathcal{A}_{S'_0}$ have sets of states Q and Q', respectively. The saturation generates the set $\{\pi_X(S_0 \,\&\, S'_0), \ldots, \pi_X(S_n \,\&\, S'_n)\}$ with $S_i \subseteq Q$, $S'_i \subseteq Q'$ for all $0 \leq i \leq n$. The size of this set is $2^{|Q|+|Q'|}$ in the worst case. In terms of the automata operations, this fixpoint expansion corresponds to first determinizing both $\mathcal{A}_{S_0}$ and $\mathcal{A}_{S'_0}$ and only then using the product construction (cf. Sect. 2.4). The automata intersection, however, works for nondeterministic automata too—the determinization is not needed. Implementing this standard product construction on terms would mean transforming the original fixpoint above into the following fixpoint with a *flattened product*: $D = \{\pi_X(S_0 \, [\&] \, S'_0)\} - \mathbf{0}^{\divideontimes}$ where $[\&]$ is the augmented product for conjunction. This way, we can decrease the worst-case size of the fixpoint to $|Q| \cdot |Q'|$. A similar reasoning holds for terms of the form $\{\pi_X(S_0 + S'_0)\} - \mathbf{0}^{\divideontimes}$. Formally, the technique can be implemented by the following pair of sub-term rewriting rules where S and S' are non-empty sets of terms:

$$S + S' \rightsquigarrow S \, [+] \, S', \tag{37}$$

$$S \mathbin{\&} S' \rightsquigarrow S \, [\&] \, S'. \tag{38}$$

Observe that for terms obtained from WS2S formulae using the translation from Sect. 3, the rules are not helpful in their given form. Consider, for instance, the term $\{\pi_X(\{r\} \mathbin{\&} \{q\})\} - \mathbf{0}^{\divideontimes}$ obtained from a formula $\exists X.\, \varphi \wedge \psi$ with φ and ψ being atoms. The term would be, using rule (38), rewritten into the term $\{\pi_X(\{r \mathbin{\&} q\})\} - \mathbf{0}^{\divideontimes}$. Then, during a subsequent fixpoint computation, we might obtain a fixpoint of the following form: $\{\pi_X(\{r \mathbin{\&} q\}), \pi_X(\{r \mathbin{\&} q, r_1 \mathbin{\&} q_1\}),$ $\pi_X(\{r_1 \mathbin{\&} q_1, r_2 \mathbin{\&} q_2\})\}$, where the occurrences of the projection π_X disallow one to perform the desired union of the inner sets, and so the application of rule (38) did not help. We therefore need to equip our procedure with a rewriting rule that can be used to push the projection inside a set term S:

$$\pi_X(S) \rightsquigarrow \{\pi_X(t) \mid t \in S\}. \tag{39}$$

In the example above, using rule (39) we would now obtain the term $\{\pi_X(r \mathbin{\&} q)\} - \mathbf{0}^{\divideontimes}$ (note that we rewrote $\{\{\cdot\}\}$ to $\{\cdot\}$ as mentioned in Sect. 3) and the fixpoint $\{\pi_X(r \mathbin{\&} q), \pi_X(r_1 \mathbin{\&} q_1),$ $\pi_X(r_2 \mathbin{\&} q_2)\}$. The correctness of the rules is guaranteed by Lemma 4.

We, however, still have to note that there is a danger related with the rules (37)–(39). Namely, if they are applied to some terms in a partially evaluated fixpoint but not to all, the form of these terms might get different (cf. $\pi_X(\{r \mathbin{\&} q\})$ and $\pi_X(r \mathbin{\&} q)$), and it will not be possible to combine them as source states of TA transitions when computing Δ_a, leading thus to an incorrect result. We resolve the situation in such a way that we apply the rules as a pre-processing step only before we start evaluating the top-level fixpoint, which ensures that all terms will subsequently be generated in a compatible form.

4.5 Nondeterministic Union

Optimization of the product term saturations from the previous section can be pushed one step further for terms of the form $\{\pi_X(S + S')\} - \mathbf{0}^{\divideontimes}$. The idea is to use the *nondeterministic TA union* to implement the union operation instead of the product construction. The TA union is implemented as the component-wise union of the two TAs. Its size is hence linear to the size of the input instead of quadratic as in the case of the product (i.e., $|Q| + |Q'|$ instead of $|Q| \cdot |Q'|$). To work correctly, the nondeterministic union requires disjoint input sets of states (otherwise, the combination of the two transition functions could generate runs that are not possible in either of the input TAs). We implement the nondeterministic union through the following rewriting rule:

$$S + S' \rightsquigarrow S \cup S' \qquad \text{for } S \not\bowtie S' \tag{40}$$

where S and S' are sets of terms (similarly to Sect. 4.4, in order to successfully reduce the fixpoint state space on terms obtained from WS2S formulae, we also need to apply rule (39) to push projection inside set terms). The relation $\not\bowtie$ used in the rule is the *non-interference* of terms, which generalizes the state space disjointness requirement of the nondeterministic union of TAs. Its complement, the *interference* of terms $\bowtie$, is defined using the following equivalences:

$$S \bowtie S' \Leftrightarrow S = S' \vee \exists t \in S, t' \in S'. t \bowtie t' \tag{41}$$

$$t \,\&\, u \bowtie t' \,\&\, u' \Leftrightarrow t \bowtie t' \vee u \bowtie u' \tag{42}$$

$$t + u \bowtie t' + u' \Leftrightarrow t \bowtie t' \vee u \bowtie u' \tag{43}$$

$$\overline{t} \bowtie \overline{t'} \Leftrightarrow t \bowtie t' \tag{44}$$

$$\pi_X(t) \bowtie \pi_X(t') \Leftrightarrow t \bowtie t' \tag{45}$$

$$D \bowtie t \Leftrightarrow D^{\mathsf{s}} \bowtie t \tag{46}$$

$$t \bowtie D \Leftrightarrow t \bowtie D^{\mathsf{s}} \tag{47}$$

$$q \bowtie r \Leftrightarrow \exists 1 \leq k \leq n. q, r \in Q_k \tag{48}$$

For terms t and u that are not matched by any rule above, we define $t \not\bowtie u$ (for instance, $t_1 \,\&\, t_2 \not\bowtie u_1 + u_2$). Interference between terms tells us when we cannot perform the rewriting. Intuitively, this happens when we obtain a term $\{S + S'\}$ where S and S' contain states from the same base automaton $\mathcal{B}_k$ with the set of states Q_k.

In order to avoid interference in the terms obtained from WS2S formulae, we can perform the following pre-processing step: When translating a WS2S formula φ into a term t_φ, we create a special version of a base TA for every occurrence of an atomic formula in φ. This way, we can never mix up terms that emerged from different subformulae to enable a transition that would otherwise stay disabled.

To use rule (40), it is necessary to modify treatment of the sink state $\emptyset$ in the definition of Δ of Sect. 3. The technical difficulty we need to circumvent is that (unlike for finite word automata) the nondeterministic union of two (even complete) TAs is not complete.

This can cause situations such as the following: let $D = \{\pi_X(\{\overline{t}\} + \{\overline{r}\})\} - \mathbf{0}^{\maltese}$ such that $\Delta_{\mathbf{0}}(t, t) = \{t\}$, $\Delta_{\mathbf{0}}(r, r) = \{r\}$, and $\mathcal{R}(t)$ and $\mathcal{R}(r)$ are both *true*, i.e., both t and r can accept any $\mathbf{0}$-tree, which also means that the union of their complements should not accept any $\mathbf{0}$-tree. Indeed, the saturation of D is the set term $D^{\mathsf{s}} = reach_{\Delta_{\mathbf{0}}}(\{\pi_X(\{\overline{t}\} + \{\overline{r}\})\}) = \{\pi_X(\{\overline{t}\} + \{\overline{r}\})\}$ where it holds that $\neg\mathcal{R}(\pi_X(\{\overline{t}\} + \{\overline{r}\}))$, i.e., it does not accept any $\mathbf{0}$-tree. On the other hand, if we use the new rule (40) together with rule (39), we obtain the term $\{\pi_X(\overline{t}), \pi_X(\overline{r})\} - \mathbf{0}^{\maltese}$. When computing its saturation, we will obtain a new element $\Delta_{\mathbf{0}}(\pi_X(\overline{t}), \pi_X(\overline{r})) = \pi_X(\overline{\emptyset})$. The term $\pi_X(\overline{\emptyset})$ was constructed using the implicit rule of Sect. 3 that sends the otherwise undefined successors of a pair of terms to $\{\emptyset\}$. Note that $\mathcal{R}(\pi_X(\overline{\emptyset}))$ is *true*, yielding that the fixpoint approximation $\{\pi_X(\overline{t}), \pi_X(\overline{r}), \pi_X(\overline{\emptyset})\}$ is a root state, so a $\mathbf{0}$-tree is accepted. Therefore, the application of the new rule (40) changed the language.

Although the previous situation cannot happen with terms obtained from WS2S formulae using the translation rules from Sect. 3, in order to formulate a correctness claim for any terms constructed using our grammar, we remedy the issue by modifying the definition of implicit transitions of Δ to $\{\emptyset\}$ from Sect. 3. We give the modified transition function $\Delta^\sharp$ in Fig. 6.

Note that in the previous example, when using the modified transition function $\Delta^\sharp$ for computing the saturation of the term $\{\pi_X(\overline{t}), \pi_X(\overline{r})\} - \mathbf{0}^{\maltese}$, we would from $t \not\bowtie r$ deduce that $\pi_X(\overline{t}) \not\bowtie \pi_X(\overline{r})$. As a consequence, $\Delta^\sharp_{\mathbf{0}}(\pi_X(\overline{t}), \pi_X(\overline{r})) = \{\emptyset\}$, which is not accepting.

We will denote the semantics of a term t obtained using $\Delta^\sharp$ instead of Δ as $\mathcal{L}^\sharp(t)$. First, we show that the properties of terms from Sect. 3 under the original semantics hold also for the modified semantics.

Fig. 6 Modified transition function

$$\Delta^\sharp_a(t,t') = \begin{cases} \Delta^\bullet_a(t,t') & \text{if } t \bowtie t' \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

$$\Delta^\bullet_a(t+u, t'+u') = \Delta^\sharp_a(t,t')\,[+]\,\Delta^\sharp_a(u,u')$$

$$\Delta^\bullet_a(t \mathbin{\&} u, t' \mathbin{\&} u') = \Delta^\sharp_a(t,t')\,[\&]\,\Delta^\sharp_a(u,u')$$

$$\Delta^\bullet_a(\pi_X(t), \pi_X(t')) = \left\{\pi_X(u) \,\middle|\, u \in \Delta^\sharp_{\pi_X(a)}(t,t')\right\}$$

$$\Delta^\bullet_a(\overline{t}, \overline{t'}) = \left\{\overline{u} \,\middle|\, u \in \Delta^\sharp_a(t,t')\right\}$$

$$\Delta^\bullet_a(S, S') = \left\{ \bigcup_{t \in S, t' \in S'} \Delta^\sharp_a(t,t') \right\}$$

$$\Delta^\bullet_a(q, r) = \delta^{\mathcal{B}}_a(q, r)$$

Lemma 5 *For automata terms A_1, A_2 and a set term S, the following equalities hold:*

$$\mathcal{L}^\sharp(\{A_1\}) = \mathcal{L}^\sharp(A_1) \tag{a}$$

$$\mathcal{L}^\sharp(\{A_1 + A_2\}) = \mathcal{L}^\sharp(A_1) \cup \mathcal{L}^\sharp(A_2) \tag{b}$$

$$\mathcal{L}^\sharp(\{A_1 \mathbin{\&} A_2\}) = \mathcal{L}^\sharp(A_1) \cap \mathcal{L}^\sharp(A_2) \tag{c}$$

$$\mathcal{L}^\sharp\left(\{\overline{A_1}\}\right) = \overline{\mathcal{L}^\sharp(A_1)} \tag{d}$$

$$\mathcal{L}^\sharp(\{\pi_X(A_1)\}) = \pi_X(\mathcal{L}^\sharp(A_1)) \tag{e}$$

$$\mathcal{L}^\sharp\left(S - \mathbf{0}^{\ast}\right) = \mathcal{L}^\sharp(S) - \mathbf{0}^{\ast} \tag{f}$$

Proof In the following proofs we abuse notation and denote by $\mathcal{A}_S$ the automaton of the term S with the altered transition function $\Delta^\sharp$.

(a): We prove the following more general form of (a):

$$\mathcal{L}^\sharp(\{A_1, \ldots, A_n\}) = \mathcal{L}^\sharp\left(\bigcup_{1 \le i \le n} A^{\mathsf{e}}_i\right) \tag{g}$$

(Again, note that all expanded terms are set terms.) Intuitively, in this proof we show that determinisation does not change the modified language of a term. Let us use $\mathcal{A}_{\bigcup A^{\mathsf{e}}_i}$ to denote the TA represented by the term $\bigcup_{1 \le i \le n} A^{\mathsf{e}}_i$. Recall that we are using the modified semantics with the altered term transition function $\Delta^\sharp$.

($\subseteq$) Let τ be a tree. It holds that $\tau \in \mathcal{L}^\sharp(\{A_1, \ldots, A_n\})$ if and only if $\tau \in \mathcal{L}^\sharp\left(\{A^{\mathsf{e}}_1, \ldots, A^{\mathsf{e}}_n\}\right)$, i.e., if there is an accepting run ρ on τ in $\mathcal{A}_{\{A^{\mathsf{e}}_1, \ldots, A^{\mathsf{e}}_n\}}$. Note that ρ maps all leaves of τ to the terms from $\{A^{\mathsf{e}}_1, \ldots, A^{\mathsf{e}}_n\}$, i.e., each leaf of τ is labelled by some A^{e}_i, which is a *set* of terms of a lower level (such a set term can be seen as a *macrostate*—i.e., a set of states—from determinisation of TAs). Since ρ is accepting, there is a term $r \in \rho(\epsilon)$ such that $\mathcal{R}(r)$. Note that because $\mathcal{R}(r)$, it follows that $r \neq \emptyset$.

We will now use ρ to construct a run ρ' of $\mathcal{A}_{\bigcup A^{\mathsf{e}}_i}$ on τ. The run ρ' will now map every position of τ to a single term. For the root position, we set $\rho'(\epsilon) = r$. We proceed by induction as follows: For all non-leaf positions $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$ such that $\rho'(w) = u$, assume that in the original run it holds that $\rho(w.\mathrm{L}) = U_\mathrm{L}$ and $\rho(w.\mathrm{R}) = U_\mathrm{R}$. Then, let $u_\mathrm{L} \in U_\mathrm{L}$ and $u_\mathrm{R} \in U_\mathrm{R}$ be terms such that $u \in \Delta^\sharp(u_\mathrm{L}, u_\mathrm{R})$ (the presence of such terms is guaranteed by (54)). The following inductive invariant holds: If $u \neq \emptyset$, then $u_{\mathtt{L}} \neq \emptyset$ and $u_{\mathtt{R}} \neq \emptyset$ (the invariant follows from (54), the fact that $r \neq \emptyset$, and (41)). We set $\rho'(w.\mathtt{L}) = u_{\mathtt{L}}$ and $\rho'(w.\mathtt{R}) = u_{\mathtt{R}}$.

As a consequence, we have that $\forall w \in \mathit{leaf}(\tau) : \rho'(w) \in \bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}$. Then, for each $w \in \mathrm{dom}(\tau)$, it holds that $\rho'(w) \in \mathit{reach}_{\Delta^\sharp}(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}})$ where $\Delta^\sharp$ is the (modified) transition function of $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$. This follows from the definition of modified transition function for set terms (54). Therefore, ρ' is a run of $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ on τ and is accepting, so $\tau \in \mathcal{L}^\sharp\left(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}\right)$.

($\supseteq$) Consider a tree $\tau \in \mathcal{L}^\sharp\left(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}\right)$. Then there is an accepting run ρ on τ in $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$. We can then use ρ to construct the run ρ' on $\mathrm{dom}(\tau)$ defined as follows: For $u \in \mathit{leaf}(\tau)$, if $\rho(u) \in A_i^{\mathsf{e}}$, we set $\rho'(u) = A_i^{\mathsf{e}}$. For $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$, we set $\rho'(w) = r$ such that $\{r\} = \Delta^\sharp_{\tau(w)}(\rho'(w.\mathtt{L}), \rho'(w.\mathtt{R}))$ (we know that $\Delta^\sharp_{\tau(w)}(\rho'(w.\mathtt{L}), \rho'(w.\mathtt{R}))$ is a singleton set due to (54)). For the constructed run ρ', it now holds that $\forall w \in \mathrm{dom}(\tau) : \rho(w) \in \rho'(w)$, therefore ρ' is an accepting run on τ in $\mathcal{A}_{\{A_1^{\mathsf{e}},\ldots,A_n^{\mathsf{e}}\}}$, i.e., $\tau \in \mathcal{L}^\sharp(\{A_1, \ldots, A_n\})$.

(*b*)–(*e*): The proof is identical to the proof of corresponding variant in Lemma 3 (with altered term transition function).

(*f*): The proof is similar to the proof of Lemma 3*f* with one exception. In particular, in the proof of (the modified version of) Claim 1, we need to make use of the fact that interference is preserved along transition relation, which is formalized in the following claim.

Claim 5 *For two terms t_1, t_2 such that $t_1 \bowtie t_2$, symbol a, and for each $t \in \Delta^\sharp_a(t_1, t_2)$ it holds that $t \bowtie t_1$ and $t \bowtie t_2$.*

Proof By induction on the structure of terms:

– *Base case*: Let t_1 and t_2 be states of some base automata. From $t_1 \bowtie t_2$ and (48), we can deduce that t_1 and t_2 are both states of some base automaton $\mathcal{B}_k$, i.e., $t_1, t_2 \in Q_k$. Then it also holds that $\Delta^\sharp_a(t_1, t_2) \subseteq Q_k$, so for every $t \in \Delta^\sharp_a(t_1, t_2)$, we have that $t \bowtie t_1$ and $t \bowtie t_2$.

Let us now continue with inductive cases.

– Let $t_1 = u_1 \,\&\, v_1$ and $t_2 = u_2 \,\&\, v_2$. From (42), it follows that either $u_1 \bowtie u_2$ or $v_1 \bowtie v_2$.

$$\begin{aligned}
\Delta^\sharp_a(t_1, t_2) &= \Delta^\sharp_a(u_1 \,\&\, v_1, u_2 \,\&\, v_2) \\
&= \Delta^\bullet_a(u_1 \,\&\, v_1, u_2 \,\&\, v_2) && \wr(49)\int \\
&= \Delta^\sharp_a(u_1, u_2)\, [\&]\, \Delta^\sharp_a(v_1, v_2) && \wr(51)\int \\
&= \{u \,\&\, v \mid u \in \Delta^\sharp_a(u_1, u_2) \wedge v \in \Delta^\sharp_a(v_1, v_2)\} && \wr\text{def. of } [\&]\int
\end{aligned}$$

Therefore, for all $t = u \,\&\, v \in \Delta^\sharp_a(t_1, t_2)$:

– if $u_1 \bowtie u_2$, then $u \bowtie u_1$ and $u \bowtie u_2$, so, from (42), it also holds that $t \bowtie t_1$ and $t \bowtie t_2$; and
– if $v_1 \bowtie v_2$, then $v \bowtie v_1$ and $v \bowtie v_2$, so, from (42), it also holds that $t \bowtie t_1$ and $t \bowtie t_2$.

– The proofs for other inductive cases are similar.

■

The other parts of the proof are similar. □

Lemma 6 *For sets of terms S and S' such that $S \neq \emptyset$ and $S' \neq \emptyset$, we have:*

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

Proof (a): $(\subseteq)$ Let $\tau \in \mathcal{L}^\sharp\left(\{S + S'\}\right)$. From Lemma 5*b* we have $\mathcal{L}^\sharp\left(\{S + S'\}\right) = \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp\left(S'\right)$. Hence there are runs ρ_1 in $\mathcal{A}_{S^{\mathsf{e}}}$ and ρ_2 in $\mathcal{A}_{S'^{\mathsf{e}}}$ on τ such that for all $w \in \mathrm{dom}(\tau)$, $\rho_1(w) \neq \emptyset \wedge \rho_2(w) \neq \emptyset$, and, moreover, at least one of them is accepting. Note that both runs exist since the transition function $\Delta^\sharp$ is complete (for a pair of terms t_1 and t_2, (i) if $t_1 \not\bowtie t_2$, then trivially $\Delta^\sharp(t_1, t_2) = \{\emptyset\} \neq \emptyset$ and (ii) if $t_1 \bowtie t_2$, then, from the definition of modified transition function we have $\Delta^\sharp(t_1, t_2) = \Delta^\bullet(t_1, t_2) \neq \emptyset$). Then, we can construct a mapping ρ from τ defined such that for all $w \in \mathrm{dom}(\tau)$, we set $\rho(w) = \rho_1(w) + \rho_2(w)$. Note that ρ is a run of $\mathcal{A}_{\{t_1^{\mathsf{e}} + t_2^{\mathsf{e}} \mid t_1 \in S, t_2 \in S'\}}$ on τ, i.e., it maps leaves of $\mathrm{dom}(\tau)$ to terms of the form $t_1^{\mathsf{e}} + t_2^{\mathsf{e}}$ for $t_1 \in S$ and $t_2 \in S'$. Moreover, ρ is accepting since at least one of the runs ρ_1 and ρ_2 is accepting. Therefore, $\tau \in \mathcal{L}^\sharp\left(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\}\right)$. From the definition of the augmented product, it follows that $\tau \in \mathcal{L}^\sharp\left(S\,[+]\,S'\right)$ and, finally, from Lemma 5*a*, we have $\tau \in \mathcal{L}^\sharp\left(\{S\,[+]\,S'\}\right)$.

$(\supseteq)$ Let $\tau \in \mathcal{L}^\sharp\left(\{S\,[+]\,S'\}\right)$. From Lemma 5*a*, we get $\tau \in \mathcal{L}^\sharp\left(S\,[+]\,S'\right)$, and from the definition of the augmented product, we obtain $\tau \in \mathcal{L}^\sharp\left(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\}\right)$. Therefore, there is an accepting run ρ on τ in $\mathcal{A}_{\{t_1^{\mathsf{e}} + t_2^{\mathsf{e}} \mid t_1 \in S, t_2 \in S'\}}$. Furthermore, let us consider the run ρ' of $\mathcal{A}_{\{S + S'\}}$ on τ (note that, due to (12), the definition of interference, and the completeness of the transition function, there is exactly one). By induction on the structure of τ, we can easily show that for all $w \in \mathrm{dom}(\tau)$, if $\rho(w) = t_1 + t_2$, then $\rho'(w) = S_1 + S_2$ such that $t_1 \in S_1$ and $t_2 \in S_2$ (the property clearly holds at leaves and is also preserved by the transition function). Let $\rho(\epsilon) = t_1^\epsilon + t_2^\epsilon$ and $\rho'(\epsilon) = S_1^\epsilon + S_2^\epsilon$. Since $\mathcal{R}(t_1^\epsilon + t_2^\epsilon)$, it also holds that $\mathcal{R}(S_1^\epsilon + S_2^\epsilon)$. Therefore, ρ' is accepting, so $\tau \in \mathcal{L}^\sharp\left(\{S + S'\}\right)$.

(b): Dual to (a).

(c): Identical to the proof of Lemma 4*c* (with the altered transition function). □

The following theorem shows that formula-to-term translation is correct even for the modified semantics.

Theorem 2 *Let φ be a WS2S formula. Then, $\mathcal{L}^\sharp\left(t_\varphi\right) = \mathcal{L}(\varphi)$.*

Proof In the proof we use the notion of expanded terms. By $t^{\mathsf{e},\Delta}$ we denote that a term t is expanded using term transition function Δ from Sect. 3.2. In the first step we prove $\mathcal{L}^\sharp\left(t_\psi\right) = \mathcal{L}\left(t_\psi\right)$ by showing that $\langle\psi\rangle^{\mathsf{e},\Delta} = \langle\psi\rangle^{\mathsf{e},\Delta^\sharp}$ for each subformula ψ of φ by induction on the structure of φ:

– $\varphi = \varphi_0$ where φ_0 is an atomic formula: Let I_{φ_0} be the set of leaf states and Q_{φ_0} set of states of a unique $\mathcal{A}_{\varphi_0}$. For each $q_1, q_2 \in Q_{\varphi_0}$ we have $q_1 \bowtie q_2$. Since I_{φ_0} is already expanded, $\langle\varphi_0\rangle^{\mathsf{e},\Delta} = \langle\varphi_0\rangle^{\mathsf{e},\Delta^\sharp}$.

– $\varphi = \psi_1 \wedge \psi_2$: We use the following equational reasoning.

$$\begin{aligned}
\langle\varphi\rangle^{\mathsf{e},\Delta} = \langle\psi_1 \wedge \psi_2\rangle^{\mathsf{e},\Delta} &= (\langle\psi_1\rangle \mathbin{\&} \langle\psi_2\rangle)^{\mathsf{e},\Delta} && \wr(25)\int \\
&= \langle\psi_1\rangle^{\mathsf{e},\Delta} \mathbin{\&} \langle\psi_2\rangle^{\mathsf{e},\Delta} && \wr\text{expansion propagation}\int \\
&= \langle\psi_1\rangle^{\mathsf{e},\Delta^\sharp} \mathbin{\&} \langle\psi_2\rangle^{\mathsf{e},\Delta^\sharp} && \wr\text{induction hypothesis}\int \\
&= (\langle\psi_1\rangle \mathbin{\&} \langle\psi_2\rangle)^{\mathsf{e},\Delta^\sharp} && \wr\text{expansion propagation}\int \\
&= \langle\varphi\rangle^{\mathsf{e},\Delta^\sharp} && \wr(25)\int
\end{aligned}$$

– $\varphi = \psi_1 \vee \psi_2$: Dual to $\psi_1 \wedge \psi_2$.
– $\varphi = \neg\psi$: We use the following equational reasoning.

$$\begin{aligned}
\langle\varphi\rangle^{\mathsf{e},\Delta} &= (\langle\overline{\psi}\rangle)^{\mathsf{e},\Delta} && \wr(27)\int \\
&= \overline{\langle\psi\rangle}^{\mathsf{e},\Delta} && \wr\text{expansion propagation}\int \\
&= \overline{\langle\psi\rangle}^{\mathsf{e},\Delta^\sharp} && \wr\text{induction hypothesis}\int \\
&= (\langle\overline{\psi}\rangle)^{\mathsf{e},\Delta^\sharp} && \wr\text{expansion propagation}\int \\
&= \langle\varphi\rangle^{\mathsf{e},\Delta^\sharp} && \wr(27)\int
\end{aligned}$$

– $\varphi = \exists X.\ \psi$: We use the following equational reasoning.

$$\begin{aligned}
\langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta} &= (\{\pi_X(\langle\psi\rangle)\} - \mathbf{0}^{*})^{\mathsf{e},\Delta} && \wr(28)\int \\
&= \left(reach_{\Delta_\mathbf{0}}(\{\pi_X(\langle\psi\rangle)\})\right)^{\mathsf{e},\Delta} && \wr(14)\int \\
&= reach_{\Delta_\mathbf{0}}\left(\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta})\}\right) && \wr\text{expansion propagation}\int \\
&= reach_{\Delta_\mathbf{0}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) && \wr\text{induction hypothesis}\int
\end{aligned}$$

From the inductive construction of $\langle\varphi\rangle$ let us now observe that for every $t_1, t_2 \in reach_{\Delta_\mathbf{0}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right)$ we have $t_1 \bowtie t_2$. This follows from the definition of interference and from the fact that for every set term S occurring in $\langle\psi\rangle$ and every $t_1, t_2 \in S$ it holds that $t_1 \bowtie t_2$. Based on the previous, we have

$$\begin{aligned}
\langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta} &= reach_{\Delta_\mathbf{0}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) \\
&= reach_{\Delta_\mathbf{0}^\sharp}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) && \wr\text{previous reasoning}\int \\
&= \langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta^\sharp} && \wr\text{expansion prop. and (28)}\int
\end{aligned}$$

Since $t_\varphi^{\mathsf{e},\Delta} = t_\varphi^{\mathsf{e},\Delta^\sharp}$ and the fact that for each $a \in \Sigma$ and $t_1, t_2 \in t_\varphi^{\mathsf{e},\Delta}$: $\Delta_a(t_1, t_2) = \Delta_a^\sharp(t_1, t_2)$, we have $\mathcal{L}\left(t_\varphi\right) = \mathcal{L}^\sharp\left(t_\varphi\right)$. Finally, from Theorem 1 we have $\mathcal{L}\left(t_\varphi\right) = \mathcal{L}\left(\varphi\right)$, which concludes the proof. □

Based on Lemmas 5, Lemma 6, and Theorem 2 we can show correctness of the nondeterministic union rule (40):

Lemma 7 *Let S and S' be sets of terms such that $S \not\bowtie S'$. Then*

$$\mathcal{L}^\sharp\left(\{S + S'\}\right) = \mathcal{L}^\sharp\left(S \cup S'\right).$$

Proof ($\subseteq$) From Lemma 5*b*, we have $\mathcal{L}^\sharp\left(\{S+S'\}\right) = \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp\left(S'\right)$. Let $\tau \in \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp\left(S'\right)$ and ρ be an accepting run on τ of either $\mathcal{A}_{S^\mathrm{e}}$ or $\mathcal{A}_{S'^\mathrm{e}}$. Therefore, ρ is an accepting run on τ also in $\mathcal{A}_{S^\mathrm{e} \cup S'^\mathrm{e}}$.

($\supseteq$) Let $\tau \in \mathcal{L}^\sharp\left(S \cup S'\right)$. For each $t_1 \in S^\mathrm{e}$ and $t_2 \in S'^\mathrm{e}$ it holds that $t_1 \not\bowtie t_2$, so we have that if $t \in \Delta_a^\sharp(t_1, t_2)$, then $t = \emptyset$. Therefore, if ρ is an accepting run of $\mathcal{A}_{S^\mathrm{e} \cup S'^\mathrm{e}}$ on τ, then ρ is an accepting run on τ in either $\mathcal{A}_{S^\mathrm{e}}$ or $\mathcal{A}_{S'^\mathrm{e}}$. Without loss of generality, suppose that ρ is an accepting run on τ of $\mathcal{A}_{S^\mathrm{e}}$ and let ρ' be the run of $\mathcal{A}_{\{S+S'\}}$ on τ (note that $\mathcal{A}_{\{S+S'\}}$ is deterministic and complete, so ρ' is unique). By induction on the structure of τ, we can easily show that for all $w \in \mathrm{dom}(\tau)$, if $\rho(w) = t_1$, then $\rho'(w) = S_1 + S_2$ such that $t_1 \in S_1$ (the property clearly holds at leaves and is also preserved by the modified transition function). Let $\rho(\epsilon) = t_1^\epsilon$ and $\rho'(\epsilon) = S_1^\epsilon + S_2^\epsilon$. Since $\mathcal{R}(t_1^\epsilon)$, it also holds that $\mathcal{R}(S_1^\epsilon + S_2^\epsilon)$. Therefore, ρ' is accepting, so $\tau \in \mathcal{L}^\sharp\left(\{S+S'\}\right)$. □

Note that although the optimization presented in this section can improve the worst-case number of reached terms, its use comes with a cost. In order to guarantee that rule (40) can be performed, we need to use a different base automaton for each atomic formula. A different base automaton can be obtained, e.g., by instantiating the automaton for a given formula every time with different names of states. The use of different base automata makes it, however, less likely that memoization avoids evaluating some function call (even though a similar one might have already been evaluated), which may significantly impact the overall performance of the decision procedure.

5 Experimental Evaluation

We have implemented the above introduced techniques (with the exception of Sect. 4.5 for the reasons described therein) in a prototype Haskell tool.[2] The base automata, hard-coded into the tool, were the TAs for the basic predicates from Sect. 2, together with automata for predicates $\mathrm{Sing}(X)$ and $X = \{p\}$ for a variable X and a fixed tree position p. As an additional optimisation, our tool uses the so-called *antiprenexing* (proposed already in [13]), which pushes quantifiers down the formula tree using the standard logical equivalences. Intuitively, antiprenexing reduces the complexity of elements within fixpoints by removing irrelevant parts outside the fixpoint.

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

We first considered various WS2S formulae on which MONA was successfully applied previously in the literature. On them, our tool is quite slower than MONA, which is not much surprising given the amount of optimisations built into MONA (for instance, for the benchmarks from [25], MONA on average took 0.1 s, while we timeouted). Next, we identified several parametric families of formulae (adapted from [13]), such as, e.g.,

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

Table 1 Experimental results over the following parametric families of formulae: (a) $\varphi_n^{pt} \equiv \forall Z_1, Z_2.\ \exists X_1, \ldots, X_n.\ edge(Z_1, X_1) \wedge \bigwedge_{i=1}^{n} edge(X_i, X_{i+1}) \wedge edge(X_n, Z_2)$ where $edge(X, Y) \equiv edge_{\mathrm{L}}(X, Y) \vee edge_{\mathrm{R}}(X, Y)$ and $edge_{\mathrm{L/R}}(X, Y) \equiv \exists Z.\ Z = S_{\mathrm{L/R}}(X) \wedge Z \subseteq Y$ (b) $\varphi_n^{cnst} \equiv \exists X.\ X = \{(\mathrm{LR})^4\} \wedge X = \{(\mathrm{LR})^n\}$ (c) $\varphi_n^{sub} = \forall X_1, \ldots, X_n\ \exists X.\ \bigwedge_{i=1}^{n-1} X_i \subseteq X \Rightarrow (X_{i+1} = S_{\mathrm{L}}(X) \vee X_{i+1} = S_{\mathrm{R}}(X))$

φ	n	Running time (sec)			# of subterms/states		
		Lazy	MONA	MONA+AP	*Lazy*	MONA	MONA+AP
φ_n^{pt}	1	0.02	0.16	0.15	149	216	216
	2	0.50	–	–	937	–	–
	3	0.83	–	–	2487	–	–
	4	34.95	–	–	8391	–	–
	5	60.94	–	–	23,827	–	–
φ_n^{cnst}	80	14.60	40.07	40.05	1146	27,913	27,913
	90	21.03	64.26	64.20	1286	32,308	32,308
	100	28.57	98.42	98.91	1426	36,258	36,258
	110	38.10	–	–	1566	–	–
	120	49.82	–	–	1706	–	–
φ_n^{sub}	3	0.01	0.00	0.00	140	92	92
	4	0.04	34.39	34.47	386	170	170
	5	0.24	–	–	981	–	–
	6	2.01	–	–	2376	–	–

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

Particularly, the columns under "Running time (sec)" give the running times (in seconds) of our tool (denoted *Lazy*), MONA, and MONA with antiprenexing (MONA +AP). The columns under "# of subterms/states" characterize the size of the generated terms and automata. Namely, for our approach, we give the number of nodes in the final term tree (with the leaves being states of the base TAs). For MONA, we give the sum of the numbers of states of all the minimal deterministic TAs constructed by MONA when evaluating the formula. The "–" sign means a timeout or that the tool ran out of memory.

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

6 Related Work

The seminal works [7,30] on the automata-logic connection were the milestones leading to what we call here the classical tree automata-based decision procedure for WS*k*S [35]. Its non-elementary worst-case complexity was proved in [33], and the work [16] presents the first implementation, restricted to WS1S, with the ambition to use heuristics to counter the high complexity. The authors of [9] provide an excellent survey of the classical results and literature related to WS*k*S and tree automata.

The tool MONA [12] implements the classical decision procedures for both WS1S and WS2S. It is still the standard tool of choice for deciding WS1S/WS*k*S formulae due to its all-around most robust performance. The efficiency of MONA stems from many optimizations, both higher-level (such as automata minimization, the encoding of first-order variables used in models, or the use of multi-terminal BDDs to encode the transition function of the automaton) as well as lower-level (e.g. optimizations of hash tables, etc.) [21,23]. The MSO(Str) logic, a dialect of WS1S, can also be decided by a similar automata-based decision procedure, implemented within, e.g., JMOSEL [36] or the symbolic finite automata framework of [10]. In particular, JMOSEL implements several optimizations (such as second-order value numbering [27]) that allow it to outperform MONA on some benchmarks (MONA also provides an MSO(Str) interface on top of the WS1S decision procedure).

The original inspiration for our work are the antichain techniques for checking universality and inclusion of finite automata [1,5,11,39] and language emptiness of alternating automata [11], which use symbolic computation together with subsumption to prune large state spaces arising from subset construction. This paper is a continuation of our work on WS1S, which started by [14], where we discussed a basic idea of generalizing the antichain techniques to a WS1S decision procedure. In [13], we then presented a complete WS1S decision procedure based on these ideas that is capable to rival MONA on already interesting benchmarks. The work in [37] presents a decision procedure that, although phrased differently, is in essence fairly similar to that of [13]. One additional feature of [37] over [13] is that it can employ laziness even when generating base automata. This feature can have a significant effect for formulae with large integer constants, such as $x = 1{,}000{,}000{,}000 \land x = 1000$. While the formula is clearly unsatisfiable, MONA constructs the base automata, which might already be too large to fit in the memory.

This paper generalizes [13] to WS2S. It is not merely a straightforward generalization of the word concepts to trees. A nontrivial transition was needed from language terms of [13], with their semantics being defined straightforwardly from the semantics of sub-terms, to tree automata terms, with the semantics defined as a language of an automaton with transitions defined inductively to the structure of the term. This change makes the reasoning and correctness proof considerably more complex, though the algorithm itself stays technically quite simple. Due to our implementation in Haskell, we can, similarly to [37], avoid constructing large base automata and only construct those parts necessary to establishing the status of input formulae.

Finally, Ganzow and Kaiser [15] developed a new decision procedure for the weak monadic second-order logic on inductive structures within their tool TOSS. Their approach completely avoids automata; instead, it is based on the Shelah's composition method. The paper reports that the Toss tool could outperform Mona on two families of WS1S formulae, one derived from Presburger arithmetics and one formula of the form that we mention in our experiments as problematic for Mona but solvable easily by Mona with antiprenexing.

Acknowledgements We thank the anonymous reviewers, both of the conference and the journal version of the paper, for their careful reading of the drafts, the spotted bugs, and the helpful comments on how to improve the exposition in this paper. This work was supported by the Czech Science Foundation project 19-24397S, the FIT BUT internal project FIT-S-20-6427, and The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science—LQ1602.

References

1. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains (on Checking language inclusion of NFAs). In: TACAS'10, *LNCS*, vol. 6015, pp. 158–174. Springer (2010)
2. Basin, D., Klarlund, N.: Automata based symbolic reasoning in hardware verification. In: CAV'98, LNCS, pp. 349–361. Springer (1998)
3. Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S systems to verify parameterized networks. In: TACAS'00, *LNCS*, vol. 1785, pp. 188–203. Springer (2000)
4. Bodeveix, J., Filali, M.: FMona: A tool for expressing validation techniques over infinite state systems. In: TACAS'00, *LNCS*, vol. 1785, pp. 204–219. Springer (2000)
5. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: CIAA'08, *LNCS*, vol. 5148, pp. 57–67. Springer (2008)
6. Bozga, M., Iosif, R., Sifakis, J.: Structural invariants for parametric verification of systems with almost linear architectures. Tech. Rep. arXiv:1902.02696 (2019)
7. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science, pp. 1–11. Stanford University Press (1962)
8. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008)
10. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: POPL'14., pp. 541–554 (2014)
11. Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: TACAS'10, *LNCS*, vol. 6015, pp. 2–22. Springer (2010)
12. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: New techniques for WS1S and WS2S. In: CAV'98. LNCS, vol. 1427, pp. 516–520. Department of Computer Science, Aarhus University, Springer, BRICS (1998)
13. Fiedor, T., Holík, L., Janků, P., Lengál, O., Vojnar, T.: Lazy automata techniques for WS1S. In: TACAS'17, *LNCS*, vol. 10205, pp. 407–425. Springer (2017)
14. Fiedor, T., Holík, L., Lengál, O., Vojnar, T.: Nested antichains for WS1S. In: TACAS'15, *LNCS*, vol. 9035. Springer (2015)
15. Ganzow, T., Kaiser, L.: New Algorithm for weak monadic second-order logic on inductive structures. In: CSL'10, *LNCS*, vol. 6247, pp. 366–380. Springer (2010)
16. Glenn, J., Gasarch, W.: Implementing WS1S via finite automata. In: Workshop on Implementing Automata, *LNCS*, vol. 1260, pp. 50–63. Springer (1996)
17. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods Syst. Des. **41**(1), 83–106 (2012)
18. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: FMCAD'10, pp. 101–109. IEEE Computer Science (2010)
19. Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Automata terms in a lazy WS*k*S decision procedure. In: Proceedings of of CADE-27, *LNCS*, vol. 11716, pp. 300–318. Springer (2019)
20. Hune, T., Sandholm, A.: A case study on using automata in control synthesis. In: FASE'00, *LNCS*, vol. 1783, pp. 349–362. Springer (2000)
21. Klarlund, N.: A theory of restrictions for logics and automata. In: CAV'99, *LNCS*, vol. 1633, pp. 406–417. Springer (1999)

22. Klarlund, N., Møller, A.: MONA Version 1.4 user manual. BRICS, Department of Computer Science, Aarhus University (2001). Notes Series NS-01-1. Available from http://www.brics.dk/mona/. Revision of BRICS NS-98-3
23. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. Int. J. Found. Comput. Sci. **13**(4), 571–586 (2002)
24. Klarlund, N., Nielsen, M., Sunesen, K.: A case study in automated verification based on trace abstractions. In: Formal System Specification, The RPC-Memory Specification Case Study, *LNCS*, vol. 1169. Springer (1996)
25. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL'11, pp. 611–622. ACM (2011)
26. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: SAS'11, *LNCS*, vol. 6887, pp. 43–59. Springer (2011)
27. Margaria, T., Steffen, B., Topnik, C.: Second-order value numbering. In: GraMoT'10, *ECEASST*, vol. 30, pp. 1–15. EASST (2010)
28. Møller, A., Schwartzbach, M.: The pointer assertion logic engine. In: PLDI'01. ACM Press (2001). Also in SIGPLAN Notices **36**(5) (2001)
29. Morawietz, F., Cornell, T.: The logic-automaton connection in linguistics. In: LACL'97, *LNAI*, vol. 1582. Springer (1997)
30. Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Trans. Am. Math. Soc. **141**, 1–35 (1969)
31. Sandholm, A., Schwartzbach, M.I.: Distributed safety controllers for web services. In: FASE'98, pp. 270–284. Springer (1998)
32. Smith, M.A., Klarlund, N.: Verification of a sliding window protocol using IOA and MONA. In: FORTE/PSTV'00, *IFIP*, vol. 183, pp. 19–34. Kluwer (2000)
33. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Fifth Annual ACM Symposium on Theory of Computing. STOC'73, pp. 1–9. ACM, New York (1973)
34. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Comput. Log. **22**(4), 33:1–33:33 (2013)
35. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Math. Syst. Theory **2**(1), 57–81 (1968)
36. Topnik, C., Wilhelm, E., Margaria, T., Steffen, B.: jMosel: a stand-alone tool and jABC plugin for M2L(Str). In: SPIN'06, *LNCS*, vol. 3925, pp. 293–298. Springer (2006)
37. Traytel, D.: A coalgebraic decision procedure for WS1S. In: 24th EACSL Annual Conference on Computer Science Logic (CSL'15). Leibniz International Proceedings in Informatics (LIPIcs), vol. 41, pp. 487–503. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)
38. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: CADE'11, *LNCS*, vol. 6803, pp. 476–491. Springer (2011)
39. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: a new algorithm for checking universality of finite automata. In: CAV'06, *LNCS*, vol. 4144, pp. 17–30. Springer (2006)
40. Wulf, M.D., Doyen, L., Maquet, N., Raskin, J.F.: Antichains: alternative algorithms for LTL satisfiability and model-checking. In: TACAS'08, *LNCS*, vol. 4693. Springer (2008)
41. Wulf, M.D., Doyen, L., Raskin, J.F.: A lattice theory for solving games of imperfect information. In: HSCC'06, *LNCS*, vol. 3927. Springer (2006)
42. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: POPL'08, pp. 349–361. ACM (2008)
43. Zhou, M., He, F., Wang, B., Gu, M., Sun, J.: Array theory of bounded elements and its applications. J. Autom. Reason. **52**(4), 379–405 (2014)

The Journal of Systems & Software 201 (2023) 111673

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A symbolic algorithm for the case-split rule in solving word constraints with extensions[☆]

Yu-Fang Chen [a], Vojtěch Havlena [b], Ondřej Lengál [b,*], Andrea Turrini [c,d]

[a] Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nangang, 11500, Taipei, Taiwan
[b] Faculty of Information Technology, Brno University of Technology, Bozetechova 2, 61200, Brno, Czech Republic
[c] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Haidian District, Zhongguancun 4# South Fourth Street, 100190, Beijing, China
[d] Institute of Intelligent Software, 221, Nansha Street West, 511458, Guangzhou, China

A R T I C L E I N F O

Article history:
Received 2 December 2021
Received in revised form 13 November 2022
Accepted 28 February 2023
Available online 5 March 2023

Keywords:
String constraints
Satisfiability modulo theories
Regular model checking
Nielsen transformation
Finite automata
Monadic second-order logic over strings

A B S T R A C T

Case split is a core proof rule in current decision procedures for the theory of string constraints. Its use is the primary cause of the state space explosion in string constraint solving, since it is the only rule that creates branches in the proof tree. Moreover, explicit handling of the *case split* rule may cause recomputation of the same tasks in multiple branches of the proof tree. In this paper, we propose a symbolic algorithm that significantly reduces such a redundancy. In particular, we encode a string constraint as a regular language and proof rules as rational transducers. This allows us to perform similar steps in the proof tree only once, alleviating the state space explosion. We also extend the encoding to handle arbitrary Boolean combinations of string constraints, length constraints, and regular constraints. In our experimental results, we validate that our technique works in many practical cases where other state-of-the-art solvers fail to provide an answer; our Python prototype implementation solved over 50% of string constraints that could not be solved by the other tools.

1. Introduction

Constraint solving is a technique used as an enabling technology in many areas of formal verification and analysis, such as symbolic execution (Cadar et al., 2006; Godefroid et al., 2005; King, 1976; Sen et al., 2013), static analysis (Wang et al., 2017; Gulwani et al., 2008), or synthesis (Gulwani et al., 2011; Osera, 2019; Knoth et al., 2019). For instance, in symbolic execution, feasibility of a path in a program is tested by creating a constraint that encodes the evolution of the values of variables on the given path and checking if it is satisfiable. Due to the features used in the analyzed programs, checking satisfiability of the constraint can be a complex task. For instance, the solver has to deal with different data types, such as Boolean, Integer, Real, or String. Theories for the first three data types are well known, widely developed, and implemented in tools, while the theory for the String data type has started to be investigated only recently in Abdulla et al. (2014), Berzish et al. (2017), Bjørner et al. (2009), Chen et al. (2019a, 2018), Holík et al. (2018), Lin and Majumdar (2018), Liang et al. (2014), Wang et al. (2016), Yu et al. (2014), Abdulla et al. (2017, 2015), Kiezun et al. (2012), Lin and Barceló (2016), Berzish et al. (2021), Reynolds et al. (2019), Blotsky et al. (2018), Stanford et al. (2021), Loring et al. (2019), Trinh et al. (2020) and Chen et al. (2019b), despite having been considered already by A. A. Markov in the late 1960s in connection with Hilbert's 10th problem (see, e.g., Matiyasevich (1968), Durnev and Zetkina (2009) and Kosovskii (1976)).

Most current decision procedures for string constraints involve the so-called *case-split* rule. This rule performs a case split with respect to the possible alignment of the variables. The case-split rule is used in most, if not all, (semi-)decision procedures for string constraints, including Makanin's algorithm in Makanin (1977), Nielsen transformation (Nielsen, 1917) (also known as the Levi's lemma Levi, 1944), and the procedures implemented in most state-of-the-art solvers such as Z3 (Bjørner et al., 2009), CVC4 (Liang et al., 2014), Z3Str3 (Berzish et al., 2017), Norn (Abdulla et al., 2014), and many more. In this paper, we will explain the general idea of our symbolic approach using the Nielsen transformation, which is the simplest of the approaches; nonetheless, we believe that the approach is applicable also to other procedures.

Consider the *word equation* $xz = yw$, the primary type of *atomic string constraints* considered in this paper, where x, z, y, and w are *word variables*. When establishing satisfiability of the word equation, the Nielsen transformation (introduced in Nielsen

☆ Editor: Earl Barr.
* Corresponding author.
E-mail addresses: yfc@iis.sinica.edu.tw (Y.-F. Chen), ihavlena@fit.vutbr.cz (V. Havlena), lengal@fit.vutbr.cz (O. Lengál), turrini@ios.ac.cn (A. Turrini).

https://doi.org/10.1016/j.jss.2023.111673
0164-1212/© 2023 Elsevier Inc. All rights reserved.

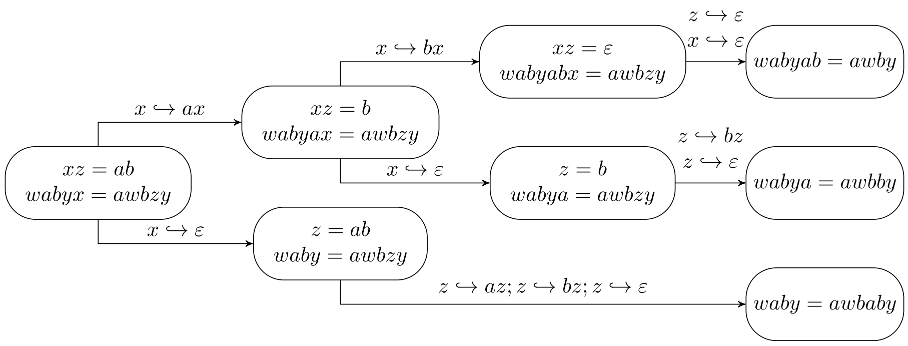


Fig. 1. A partial proof tree of applying the Nielsen transformation on the string constraint $xz = ab \wedge wabyx = awbzy$. The leaves are the outcome of completely processing the first word equation $xz = ab$. Branches leading to contradictions are omitted.

(1917)) proceeds by first performing a case split based on the possible alignments of the variables x and y, the first symbol of the left and right-hand sides of the equation, respectively. More precisely, it reduces the satisfiability problem for $xz = yw$ into satisfiability of (at least) one of the following four cases (1) y is a prefix of x, (2) x is a prefix of y, (3) x is an empty string, and (4) y is an empty string. Note that these cases are not disjoint: for instance, the empty string is a prefix of every variable. For these cases, the Nielsen transformation generates the following equations.

For the case (1), i.e., y is a prefix of x, all occurrences of x in $xz = yw$ are replaced with yx', where x' is a fresh word variable (we denote this case as $x \hookrightarrow yx'$), i.e., we obtain the equation $yx'z = yw$, which can be simplified to $x'z = w$. In fact, since the transformation $x \hookrightarrow yx'$ removes all occurrences of the variable x, we can just reuse the variable x and perform the transformation $x \hookrightarrow yx$ instead (and take this into account when constructing a model later).

Case (2) of the Nielsen transformation is just a symmetric counterpart of case (1) discussed above. For cases (3) and (4), x and y, respectively, are replaced by empty strings. Taking into account all four possible transformations of the equation $xz = yw$, we obtain the following equations:

$$(1)\ xz = w \quad (2)\ z = yw \quad (3)\ z = yw \quad (4)\ xz = w$$

(Note that the results for (1) and (4) coincide, as well as the results for (2) and (4).) If $xz = yw$ has a solution, then at least one of the above equations has a solution, too. The Nielsen transformation keeps applying the transformation rules on the obtained equations, building a proof tree and searching for a tautology of the form $\epsilon = \epsilon$.

Treating each of the obtained equations separately can cause some redundancy (as we could already see above). Let us consider the example in Fig. 1, where we apply the Nielsen transformation to solve the string constraint $xz = ab \wedge wabyx = awbzy$, where x, z, w, and y are word variables and a and b are constant symbols. After processing the first word equation $xz = ab$, we obtain a proof tree with three very similar leaf nodes $wabyab = awby$, $wabya = awbby$, and $waby = awbaby$, which share the prefixes $waby$ and awb on the left and right-hand side of the equations, respectively. If we continue applying the Nielsen transformation on the three leaf nodes, we will create three very similar subtrees, with almost identical operations. In particular, the nodes near the root of such subtrees, which transform $waby\ldots = awb\ldots$, are going to be essentially the same. The resulting proof trees will therefore start to differ only after processing such a common part. Therefore, handling those equations separately will cause that some operations will be performed multiple times. If the proof tree of each word equation has n leaves and the string constraint is a conjunction of k word equations, we might need to create n^k similar subtrees.

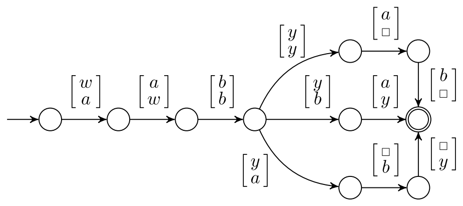

$\begin{bmatrix} w \\ a \end{bmatrix}$
$\begin{bmatrix} a \\ w \end{bmatrix}$
$\begin{bmatrix} b \\ b \end{bmatrix}$
$\begin{bmatrix} y \\ y \end{bmatrix}$
$\begin{bmatrix} a \\ \square \end{bmatrix}$
$\begin{bmatrix} b \\ \square \end{bmatrix}$
$\begin{bmatrix} y \\ b \end{bmatrix}$
$\begin{bmatrix} a \\ y \end{bmatrix}$
$\begin{bmatrix} y \\ a \end{bmatrix}$
$\begin{bmatrix} \square \\ b \end{bmatrix}$
$\begin{bmatrix} \square \\ y \end{bmatrix}$

Fig. 2. A finite automaton encoding the three equations $wabyab = awby$, $wabya = awbby$, and $waby = awbaby$.

The case split can be performed more efficiently if we process the common part of the said leaves together using a symbolic encoding. In this paper, we use an encoding of a set of equations as a regular language, which is represented by a *finite automaton*. An example is given in Fig. 2, which shows a finite automaton over a 2-track alphabet, where each of the two tracks represents one side of the equation. For instance, the equation $wabyab = awby$ is represented by the word $\begin{bmatrix} w \\ a \end{bmatrix}\begin{bmatrix} a \\ w \end{bmatrix}\begin{bmatrix} b \\ b \end{bmatrix}\begin{bmatrix} y \\ y \end{bmatrix}\begin{bmatrix} a \\ \square \end{bmatrix}\begin{bmatrix} b \\ \square \end{bmatrix}$ accepted by the automaton, where the $\square$ symbol is a padding used to make sure that both tracks are of the same length.

Given our regular language-based symbolic encoding, we need a mechanism to perform the Nielsen transformation steps on a set of equations encoded as a regular language. We show that the transformations can be encoded as *rational relations*, represented using *finite transducers*, and the whole satisfiability checking problem can be encoded within the framework of *regular model checking*. We will provide more details on how this is done in Sections 3–6 stepwise. In Section 3, we describe the approach for a simpler case where the input is a *quadratic word equation*, i.e., a word equation with at most two occurrences of every variable. In this case, the Nielsen transformation is sound and complete, that is, the solution it returns is correct and it returns a solution whenever a solution exists. In Section 4, we extend the technique to support the *conjunction* of *non-quadratic* word equations. In Section 5, we extend our approach to support arbitrary Boolean combination of string constraints. Section 6 extends our framework with two additional types of atomic string constraints—*length* and *regular constraints*—which can constrain the length of values assigned to word variables and their membership in a regular language, respectively.

We have implemented our approach in a prototype Python tool called RETRO and evaluated its performance on two benchmark sets: Kepler$_{22}$ obtained from Le and He (2018) and PYEX-HARD obtained by running the PyEx symbolic execution engine on Python programs from Reynolds et al. (2017) and collecting examples on which CVC4 or Z3 fail. RETRO solved most of the problems in Kepler$_{22}$ (on which CVC4 and Z3 do not perform well). Moreover, it solved over 50% of the benchmarks in PYEX-HARD that could be solved by neither CVC4 nor Z3.

This paper is an extended version of the paper that appeared in the proceedings of APLAS'20 (Chen et al., 2020), containing complete proofs of the presented lemmas and theorems and further extending the presented technique to handle (i) arbitrary Boolean combination of string constraints (Section 5), (ii) length constraints (Section 6.1), and (iii) regular constraints (Section 6.2).

2. Preliminaries

An *alphabet* Σ is a finite set of *characters* and a *word* over Σ is a sequence $w = a_1 \ldots a_n$ of characters from Σ, with ϵ denoting the *empty word*. We use $w_1.w_2$ (and often just w_1w_2) to denote the *concatenation* of words w_1 and w_2. Σ^* is the set of all words over Σ, $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$, and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. A *language* over Σ is a subset L of Σ^*. Given a word $w = a_1 \ldots a_n$, we use $|w|$ to denote the length n of w and $|w|_a$ to denote the number of occurrences of the character $a \in \Sigma$ in w. Further, we use $w[i]$ to denote a_i, the ith character of w, and $w[i:]$ to denote the word $a_i \ldots a_n$. When $i > n$, the value of $w[i]$ and $w[i:]$ is in both cases $\perp$, a special *undefined* value, which is different from all other values and also from itself (i.e., $\perp \neq \perp$). We use Σ^k for $k \geq 2$ to denote the *stacked alphabet* consisting of k-tuples of symbols from Σ, e.g., $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] \in \Sigma^2$ for $a, b \in \Sigma$.

2.1. Automata and transducers

A *(finite) k-tape transducer* is a quintuple $\mathcal{T} = (Q, \Sigma, \Delta, Q_i, Q_f)$ such that Q is a finite set of *states*, Σ is an alphabet, $\Delta \subseteq Q \times \Sigma_\epsilon^k \times Q$ is a set of *transitions* of the form $q \xrightarrow{a^1,\ldots,a^k} s$ for $a^1, \ldots, a^k \in \Sigma_\epsilon$, $Q_i \subseteq Q$ is a set of *initial states*, and $Q_f \subseteq Q$ is a set of *final states*. A run π of $\mathcal{T}$ over a k-tuple of words $(w_1, \ldots, w_k)$ is a sequence of transitions $q_0 \xrightarrow{a_1^1,\ldots,a_1^k} q_1$, $q_1 \xrightarrow{a_2^1,\ldots,a_2^k} q_2$, $\ldots$, $q_{n-1} \xrightarrow{a_n^1,\ldots,a_n^k} q_n \in \Delta$ such that for each $i \in [1, k]$ we have $w_i = a_1^i a_2^i \ldots a_n^i$ (note that a_m^i can be ϵ, so w_i and w_j may be of a different length, for $i \neq j$). The run π is *accepting* if $q_0 \in Q_i$ and $q_n \in Q_f$, and a k-tuple $(w_1, \ldots, w_k)$ is *accepted* by $\mathcal{T}$ if there exists an accepting run of $\mathcal{T}$ over $(w_1, \ldots, w_k)$. The *language* $\mathcal{L}(\mathcal{T})$ of $\mathcal{T}$ is defined as the k-ary relation $\mathcal{L}(\mathcal{T}) = \{(w_1, \ldots, w_k) \in (\Sigma^*)^k \mid (w_1, \ldots, w_k) \text{ is accepted by } \mathcal{T}\}$. We call the class of relations accepted by transducers *rational relations*. $\mathcal{T}$ is *length-preserving* if no transition in Δ contains ϵ; the class of relations accepted by length-preserving transducers is named as *regular relations*. For a 2-tape transducer $\mathcal{T}$ and $(w_1, w_2) \in \mathcal{L}(\mathcal{T})$, we denote w_1 as an input and w_2 as an output of $\mathcal{T}$. A *finite automaton* (FA) is a 1-tape finite transducer; languages accepted by finite automata are called *regular languages*. See, e.g., Pin (2021) for more details on automata and transducers.

Given two k-ary relations R_1, R_2, we define their *concatenation* $R_1.R_2 = \{(u_1v_1, \ldots, u_kv_k) \in (\Sigma^*)^k \mid (u_1, \ldots, u_k) \in R_1 \wedge (v_1, \ldots, v_k) \in R_2\}$ and given two binary relations R_1, R_2, we define their *composition* $R_1 \circ R_2 = \{(x, z) \in (\Sigma^*)^2 \mid \exists y \in \Sigma^* : (x, y) \in R_2 \wedge (y, z) \in R_1\}$. Given a k-ary relation R we define $R^0 = \{\epsilon\}^k$, $R^{i+1} = R.R^i$ for $i \geq 0$. *Iteration* of R is then defined as $R^* = \bigcup_{i \geq 0} R^i$. Given a language $\mathcal{L} \subseteq \Sigma^*$ and a binary relation R, we use $R(\mathcal{L})$ to denote the language $\{y \in \Sigma^* \mid \exists x \in \mathcal{L} : (x, y) \in R\}$, called the *R-image* of $\mathcal{L}$. We also use $R^{-1}(w)$ to denote the language $\{u \mid (w, u) \in R\}$, called the *preimage* of a word w.

Proposition 1 (Berstel, 1979)**.** *The following propositions hold:*

(i) The class of binary rational relations is closed under (finite) union, composition, concatenation, and iteration; it is not closed under intersection and complement.
(ii) For a binary rational relation R, a regular language $\mathcal{L}$, and a word w, the languages $R(\mathcal{L})$ and $R^{-1}(w)$ are also effectively regular (i.e., they can be computed).
(iii) The class of regular relations is closed under Boolean operations.

2.2. String constraints

Let Σ be an alphabet and $\mathbb{X}$ be a set of *word variables* ranging over Σ^* s.t. $\mathbb{X} \cap \Sigma = \emptyset$. We use $\Sigma_\mathbb{X}$ to denote the extended alphabet $\Sigma \cup \mathbb{X}$. An *assignment of* $\mathbb{X}$ is a mapping $I : \mathbb{X} \to \Sigma^*$. A *word term* is a string over the alphabet $\Sigma_\mathbb{X}$. We lift an assignment I to word terms by defining $I(\epsilon) = \epsilon$, $I(a) = a$, and $I(x.w) = I(x).I(w)$, for $a \in \Sigma$, $x \in \Sigma_\mathbb{X}$, and $w \in \Sigma_\mathbb{X}^*$. A *word equation* φ_e is of the form $t_1 = t_2$ where t_1 and t_2 are word terms. I is a *model* of φ_e if $I(t_1) = I(t_2)$. We call a word equation an *atomic string constraint*. A *string constraint* is obtained from atomic string constraints using Boolean connectives ($\wedge, \vee, \neg$), with the semantics defined in the standard manner. A string constraint is *satisfiable* if it has a model. Given a word term $t \in \Sigma_\mathbb{X}^*$, a variable $x \in \mathbb{X}$, and a word term $u \in \Sigma_\mathbb{X}^*$, we use $t[x \mapsto u]$ to denote the word term obtained from t by replacing all occurrences of x by u, e.g. $(abxcxy)[x \mapsto cy] = abcyccyy$. We call a string constraint ψ *quadratic* if each variable has at most two occurrences in ψ, and *cubic* if each variable has at most three occurrences in ψ.

We use the following terminology. Let Φ be a string constraint. A (semi-)algoritm **A** for solving string constraints is

1. *sound* if it holds that if **A** returns an assignment I to the variables of Φ, then I is a model of Φ,
2. *complete* if it holds that if Φ is satisfiable, then **A** returns a model of Φ in a finite number of steps, and
3. *terminating* if it holds that **A** always returns an assignment or false in a finite number of steps.

2.3. Monadic second-order logic on strings (MSO(STR))

We define *monadic second-order logic on strings* (MSO(Str)) (Büchi, 1960) over the alphabet Γ as follows. Let $\mathbb{W}$ be a countable set of *string variables* whose values range over Γ^* and $\mathbb{P}$ be a countable set of *set (second-order) position variables* whose values range over finite subsets of $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$ such that $\mathbb{W} \cap \mathbb{P} = \emptyset$. A formula φ of MSO(Str) is defined as

$$\begin{aligned}\varphi ::= {} & P \subseteq R \mid P = R + 1 \mid w[P] = a \mid \\ & \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \forall^{\mathbb{P}} P(\varphi) \mid \forall^{\mathbb{W}} w(\varphi)\end{aligned}$$

where $P, R \in \mathbb{P}$, $w \in \mathbb{W}$, and $a \in \Gamma$. We use $\varphi(w_1, \ldots, w_k)$ to denote that the free variables of φ are contained in $\{w_1, \ldots, w_k\}$.

The semantics of MSO(Str) is defined in Fig. 3. An MSO(Str) *variable assignment* is an assignment $\sigma : \mathbb{W} \cup \mathbb{P} \to (\Gamma^* \cup 2^{\mathbb{N}_1})$ that respects the types of variables with the additional requirement that for every $u, v \in \mathbb{W}$ we have $|\sigma(u)| = |\sigma(v)|$. (We often omit unused variables in σ.) We use $|\sigma|$ to denote the value $|\sigma(w)|$ of any $w \in \mathbb{W}$. Note that $|\sigma|$ is well defined since we assume that all $w \in \mathbb{W}$ are mapped to strings of the same length. The notation $\sigma[x \mapsto v]$ denotes a variant of σ where the assignment of variable x is changed to the value v.

We call an MSO(Str) formula a *string formula* if it contains no free position variables. Such a formula (with k free string variables) denotes a k-ary relation over Γ^*. In particular, given an MSO(Str) string formula $\varphi(w_1, \ldots, w_k)$ with k free string variables $w_1, \ldots, w_k$, we use $\mathcal{L}(\varphi)$ to denote the relation $\{(x_1, \ldots, x_k) \in (\Gamma^*)^k \mid \{w_1 \mapsto x_1, \ldots, w_k \mapsto x_k\} \models \varphi\}$. In the special case of $k = 1$, φ denotes a language $\mathcal{L}(\varphi) \subseteq \Gamma^*$.

Proposition 2 (Thatcher and Wright, 1968)**.** *The class of languages denoted by* MSO(Str) *string formulae with* 1 *free string variable is exactly the class of regular languages. Furthermore, the class of relations denoted by* MSO(Str) *string formulae with k free string variables, for $k > 1$, is exactly the class of regular relations.*

$$\begin{array}{ll}
\sigma \models P \subseteq R & \text{iff } \sigma(P) \text{ is a subset of } \sigma(R) \\
\sigma \models P = R+1 & \text{iff } \sigma(P) = \{r+1 \mid r \in \sigma(R) \text{ and } r+1 \leq |\sigma|\} \\
\sigma \models w[P] = a & \text{iff for all } p \in P \text{ it holds that } \sigma(w)[p] \text{ is } a \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff not } \sigma \models \varphi \\
\sigma \models \forall^{\mathbb{P}} P(\varphi) & \text{iff for all } v \subseteq \{1,\ldots,|\sigma|\} \text{ it holds that } \sigma[P \mapsto v] \models \varphi \\
\sigma \models \forall^{\mathbb{W}} w(\varphi) & \text{iff for all } v \in \Gamma^{|\sigma|} \text{ it holds that } \sigma[w \mapsto v] \models \varphi
\end{array}$$

Fig. 3. Semantics of MSO(Str).

$$\begin{array}{rcl@{\qquad}rcl}
\varphi \vee \psi & \triangleq & \neg(\neg\varphi \wedge \neg\psi) & \exists^{\mathbb{P}} P(\varphi) & \triangleq & \neg\forall^{\mathbb{P}} P(\neg\varphi) \\
\varphi \to \psi & \triangleq & \neg\varphi \vee \psi & \exists^{\mathbb{W}} w(\varphi) & \triangleq & \neg\forall^{\mathbb{W}} w(\neg\varphi) \\
P = R & \triangleq & P \subseteq R \;\wedge\; R \subseteq P & P = \emptyset & \triangleq & \forall^{\mathbb{P}} R(P \subseteq R)
\end{array}$$

$$\begin{array}{rcl}
Sing(P) & \triangleq & \neg(P = \emptyset) \;\wedge\; \forall^{\mathbb{P}} R(R \subseteq P \to (R = \emptyset \;\vee\; R = P)) \\
p \in R & \triangleq & Sing(p) \wedge p \subseteq R \\
p \leq r & \triangleq & \forall^{\mathbb{P}} T((p \in T \wedge \forall^{\mathbb{P}} u(u \in T \to \exists^{\mathbb{P}} v(v = u+1 \wedge v \in T))) \to r \in T) \\
p < r & \triangleq & p \leq r \wedge \neg(p = r) \\
x = 1 & \triangleq & \forall^{\mathbb{P}} u(u \leq x \to u = x) \qquad x = \$ \;\triangleq\; \forall^{\mathbb{P}} u(x \leq u \to u = x) \\
w_1[P] = w_2[R] & \triangleq & \bigvee_{a \in \Gamma} (w_1[P] = a \wedge w_2[R] = a)
\end{array}$$

for all $j \geq 1$ and any term γ:

$$\begin{array}{rcl}
P = R + j & \triangleq & \exists^{\mathbb{P}} S_1 \ldots \exists^{\mathbb{P}} S_j (S_1 = P+1 \wedge S_2 = S_1 + 1 \wedge \ldots \wedge R = S_j + 1) \\
p = j & \triangleq & \exists^{\mathbb{P}} z(z = 1 \wedge p = z + (j-1)) \\
w[P+j] = \gamma & \triangleq & \exists^{\mathbb{P}} S(S = P + j \wedge w[S] = \gamma) \\
w[P-j] = \gamma & \triangleq & \exists^{\mathbb{P}} S(P = S + j \wedge w[S] = \gamma) \\
w[\$-j] = \gamma & \triangleq & \exists^{\mathbb{P}} r \exists^{\mathbb{P}} s(r = \$ \wedge r = s + j \wedge w[s] = \gamma) \\
w[j] = \gamma & \triangleq & \exists^{\mathbb{P}} p(p = j \wedge w[p] = \gamma) \\
p \geq j & \triangleq & \exists^{\mathbb{P}} s(s = j \wedge p \geq s)
\end{array}$$

Fig. 4. Syntactic sugar for MSO(Str).

Syntactic sugar for MSO(STR). In Fig. 4, we define the standard syntactic sugar to allow us to write more concise MSO(Str) formulae. Most of the sugar is standard, let us, however, explain some of the less standard notation: $Sing(P)$ denotes that P is a singleton set of positions, $p \leq r$ denotes that p and r are single positions and that p is less than or equal to r, $x = 1$ and $x = \$$ denote that x is the first and the last position respectively, and $P = R + j$ denotes that P is equal to R with all positions incremented by j. We also extend our syntax to allow first-order variables (we abuse notation and use the same quantifier notation as for second-order variables, but denote the first-order variable with a lowercase letter):

$$\forall^{\mathbb{P}} p(\varphi) \triangleq \forall^{\mathbb{P}} P(Sing(P) \to \varphi[p \mapsto P])$$
$$\exists^{\mathbb{P}} p(\varphi) \triangleq \exists^{\mathbb{P}} P(Sing(P) \wedge \varphi[p \mapsto P])$$

where $\varphi[p \mapsto P]$ denotes the substitution of all free occurrences of p in φ by P.

2.4. Nielsen transformation

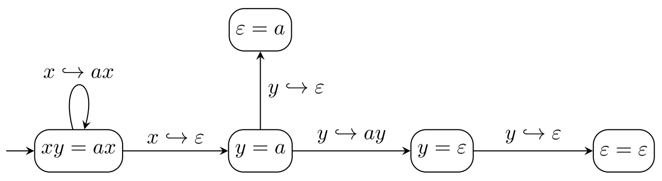
As already briefly mentioned in the introduction, the Nielsen transformation can be used to check satisfiability of a conjunction of word equations. We use the three rules shown in Fig. 5; besides the rules $x \hookrightarrow \alpha x$ and $x \hookrightarrow \epsilon$ that we have seen in the introduction, there is also the (trim) rule, used to remove a shared prefix from both sides of the equation.

$$\frac{\alpha u = \alpha v}{u = v}\;(\text{trim}) \qquad \frac{xu = v}{u[x \mapsto \varepsilon] = v[x \mapsto \varepsilon]}\;(x \hookrightarrow \varepsilon)$$

$$\frac{xu = \alpha v}{x(u[x \mapsto \alpha x]) = v[x \mapsto \alpha x]}\;(x \hookrightarrow \alpha x)$$

Fig. 5. Rules of the Nielsen transformation, with $x \in \mathbb{X}$, $\alpha \in \Sigma_{\mathbb{X}}$, and $u, v \in \Sigma_{\mathbb{X}}^*$. Symmetric rules are omitted.

Given a system of word equations, multiple Nielsen transformations might be applicable to it, resulting in different transformed equations on which other Nielsen transformations can be performed, as shown in Fig. 1. Trying all possible transformations generates a tree (or a graph in general) whose nodes contain conjunctions of word equations and whose edges are labeled with the applied transformation. The conjunction of word equations in the root of the tree is satisfiable if and only if at least one of the



$x \hookrightarrow ax$
$xy = ax$
$x \hookrightarrow \varepsilon$
$y = a$
$y \hookrightarrow \varepsilon$
$\varepsilon = a$
$y \hookrightarrow ay$
$y = \varepsilon$
$y \hookrightarrow \varepsilon$
$\varepsilon = \varepsilon$

Fig. 6. Proof graph of the equation $xy = ax$ generated by the Nielsen transformation.

leaves in the graph is a tautology, i.e., it contains a conjunction of the form $\epsilon = \epsilon \wedge \cdots \wedge \epsilon = \epsilon$. As an example, consider the satisfiable equation $xy = ax$ where x, y are word variables and a is a symbol with the proof graph in Fig. 6.

Lemma 3 (*cf.* Makanin, 1977; Diekert, 2002)**.** *The Nielsen transformation is sound and complete. Moreover, if the system of word equations is quadratic, the proof graph is finite.*

Lemma 3 is correct even if we construct the proof tree using the following strategy: every application of $x \hookrightarrow \alpha x$ or $x \hookrightarrow \epsilon$ is followed by as many applications of the (trim) rule as possible. We use $x \rightarrowtail \alpha x$ to denote the application of one $x \hookrightarrow \alpha x$ rule followed by as many applications of (trim) as possible, and $x \rightarrowtail \epsilon$ for the application of $x \hookrightarrow \epsilon$ followed by (trim).

2.5. Regular model checking

Regular model checking (RMC) (cf. Kesten et al. (2001), Wolper and Boigelot (1998), Bouajjani et al. (2000), Abdulla (2012) and Bouajjani et al. (2012)) is a framework for verifying infinite state systems. In RMC, each *system configuration* is represented as a word over an alphabet Σ. The set of *initial configurations* $\mathcal{I}$ and *destination configurations* $\mathcal{D}$ are captured as regular languages over Σ. The *transition relation* $\mathcal{T}$ is captured as a binary rational relation over Σ^*. A regular model checking *reachability problem* is represented by the triplet $(\mathcal{I}, \mathcal{T}, \mathcal{D})$ and asks whether $\mathcal{T}^{rt}(\mathcal{I}) \cap \mathcal{D} \neq \emptyset$, where $\mathcal{T}^{rt}$ represents the reflexive and transitive closure of $\mathcal{T}$. One way how to solve the problem is to start computing the sequence $\mathcal{T}^{(0)}(\mathcal{I}), \mathcal{T}^{(1)}(\mathcal{I}), \mathcal{T}^{(2)}(\mathcal{I}), \ldots$ where $\mathcal{T}^{(0)}(\mathcal{I}) = \mathcal{I}$ and $\mathcal{T}^{(n+1)}(\mathcal{I}) = \mathcal{T}(\mathcal{T}^{(n)}(\mathcal{I}))$. During the computation of the sequence, we can check whether we find $\mathcal{T}^{(i)}(\mathcal{I})$ that overlaps with $\mathcal{D}$, and if yes, we can deduce that $\mathcal{D}$ is reachable. On the other hand, if we obtain a sequence such that $\bigcup_{0 \le i < n} \mathcal{T}^i(\mathcal{I}) \supseteq \mathcal{T}^n(\mathcal{I})$, we know that we have explored all possible system configurations without reaching $\mathcal{D}$, so $\mathcal{D}$ is unreachable. The RMC reachability problem is in general *undecidable* (this can be easily shown, e.g., by a reduction from Turing machine configuration reachability).

3. Solving word equations using RMC

In this section, we describe a symbolic RMC-based framework for solving string constraints. The framework is based on encoding a string constraint into a regular language and encoding steps of the Nielsen transformation as a rational relation. Satisfiability of a string constraint is then reduced to a reachability problem of RMC.

3.1. Nielsen transformation as word operations

In the following, we describe how the Nielsen transformation of a single word equation can be expressed as operations on words. We view a word equation $eq\colon \mathsf{t}_\ell = \mathsf{t}_r$ as a pair of word terms $e_{eq} = (\mathsf{t}_\ell, \mathsf{t}_r)$ corresponding to the left and right hand sides of the equation respectively; therefore $e_{eq} \in \Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*$. Without loss of generality we assume that $\mathsf{t}_\ell[1] \neq \mathsf{t}_r[1]$; if this is not the case, we pre-process the equation by applying the (trim) Nielsen transformation (cf. Fig. 5) to trim the common prefix of t_ℓ and t_r.

Example 1. The word equation $eq_1\colon xay = yx$ is represented by the pair of word terms $e_1 = (xay, yx)$. The full proof graph generated by applying the Nielsen transformation is depicted in Fig. 7. □

A rule of the Nielsen transformation (cf. Section 2.4) is represented using a (partial) function $\tau\colon (\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*) \to (\Sigma_{\mathbb{X}}^* \times \Sigma_{\mathbb{X}}^*)$. Given a pair of word terms $(\mathsf{t}_\ell, \mathsf{t}_r)$ of a word equation eq, the function τ transforms it into a pair of word terms of a word equation eq' that would be obtained by performing the corresponding step of the Nielsen transformation on eq. Before we express the rules of the Nielsen transformation, we define functions performing the corresponding substitution. For $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$ we define

$$\begin{aligned}\tau_{x\mapsto\alpha x} &= \{\, (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto (\mathsf{t}'_\ell, \mathsf{t}'_r) \mid \mathsf{t}'_\ell = \mathsf{t}_\ell[x \mapsto \alpha x] \wedge \mathsf{t}'_r = \mathsf{t}_r[x \mapsto \alpha x] \,\} \text{ and} \\ \tau_{x\mapsto\epsilon} &= \{\, (\mathsf{t}_\ell, \mathsf{t}_r) \mapsto (\mathsf{t}'_\ell, \mathsf{t}'_r) \mid \mathsf{t}'_\ell = \mathsf{t}_\ell[x \mapsto \epsilon] \wedge \mathsf{t}'_r = \mathsf{t}_r[x \mapsto \epsilon] \,\}.\end{aligned} \tag{1}$$

The function $\tau_{x\mapsto\alpha x}$ performs a substitution $x \mapsto \alpha x$ while the function $\tau_{x\mapsto\epsilon}$ performs a substitution $x \mapsto \epsilon$.

Example 2. Consider the pair of word terms $e_1 = (xay, yx)$ from Example 1. The application $\tau_{x\mapsto yx}(e_1)$ would produce the pair $e_2 = (yxay, yyx)$ and the application $\tau_{x\mapsto\epsilon}(e_1)$ would produce the pair $e_3 = (ay, y)$. □

The functions introduced above do not take into account the first symbols of each side and do not remove a common prefix of the two sides of the equation, which is a necessary operation for the Nielsen transformation to terminate. Let us, therefore, define

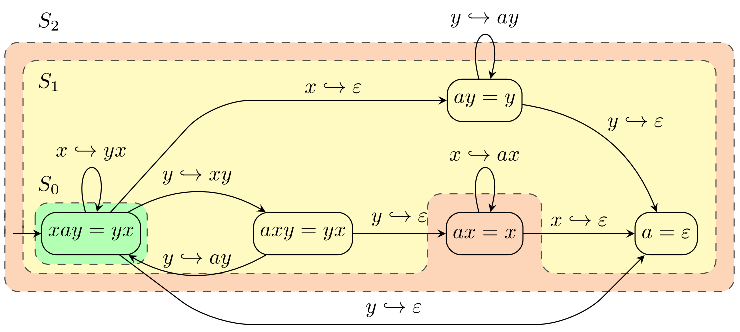


Fig. 7. Proof graph for a run of the Nielsen transformation on the equation $xay = yx$. The sets S_0, S_1, and S_2 are the sets of nodes explored in 0, 1, and 2 steps of our algorithm, respectively.

the following function, which trims (the longest) matching prefix of word terms of the two sides of an equation:

$$\tau_{trim} = \{\, (\mathrm{t}_\ell, \mathrm{t}_r) \mapsto (\mathrm{t}'_\ell, \mathrm{t}'_r) \mid \exists i \geq 1 \forall j < i \,\big(\, \mathrm{t}_\ell[i] \neq \mathrm{t}_r[i] \wedge \mathrm{t}_\ell[j] = \mathrm{t}_r[j] \wedge \mathrm{t}'_\ell = \mathrm{t}_\ell[i{:}] \wedge \mathrm{t}'_r = \mathrm{t}_r[i{:}] \,\big)\, \}. \tag{2}$$

Example 3. Continuing in our running example, the application $\tau_{trim}(e_2)$ produces the pair $e'_2 = (xay, yx)$ and, furthermore, $\tau_{trim}(e_3)$ produces the pair $e'_3 = (ay, y)$. □

Now we are ready to define functions corresponding to the rules of the Nielsen transformation. In particular, the rule $x \rightarrowtail \alpha x$ and its symmetric variant (i.e., x is the first symbol of either left or right side of an equation) for $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$ (cf. Section 2.4) can be expressed using the function

$$\tau_{x \rightarrowtail \alpha x} = \tau_{trim} \circ \{\, (\mathrm{t}_\ell, \mathrm{t}_r) \mapsto \tau_{x \mapsto \alpha x}(\mathrm{t}_\ell, \mathrm{t}_r) \mid (\mathrm{t}_r[1] = x \wedge \mathrm{t}_\ell[1] = \alpha) \vee (\mathrm{t}_r[1] = \alpha \wedge \mathrm{t}_\ell[1] = x) \,\} \tag{3}$$

while the rule $x \rightarrowtail \epsilon$ and its symmetric variant for $x \in \mathbb{X}$ can be expressed as the function

$$\tau_{x \rightarrowtail \epsilon} = \tau_{trim} \circ \{\, (\mathrm{t}_\ell, \mathrm{t}_r) \mapsto \tau_{x \mapsto \epsilon}(\mathrm{t}_\ell, \mathrm{t}_r) \mid (\mathrm{t}_\ell[1] = x \vee \mathrm{t}_r[1] = x)\}. \tag{4}$$

If we keep applying the functions defined above on individual pairs of word terms, while searching for the pair (ϵ, ϵ)—which represents the case when a solution to the original equation eq exists—, we would obtain the Nielsen transformation graph (cf. Section 2.4). In the following, we show how to perform the steps symbolically on a representation of a *set* of word equations at once.

3.2. Symbolic algorithm for word equations

In this section, we describe the main idea of our symbolic algorithm for solving word equations. We first focus on the case of a single word equation and in subsequent sections extend the algorithm to a richer class.

Our algorithm is based on applying the transformation rules not on a single equation, but on a whole *set of equations* at once. For this, we define the relations $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \epsilon}$ that aggregate the versions of $\tau_{x \rightarrowtail \alpha x}$ and $\tau_{x \rightarrowtail \epsilon}$ for all possible $x \in \mathbb{X}$ and $\alpha \in \Sigma_{\mathbb{X}}$. The signature of these relations is $(\Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}) \times (\Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}})$ and they are defined as follows:

$$\mathcal{T}_{x \rightarrowtail \alpha x} = \bigcup_{y \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} \tau_{y \rightarrowtail \alpha y} \qquad \mathcal{T}_{x \rightarrowtail \epsilon} = \bigcup_{y \in \mathbb{X}} \tau_{y \rightarrowtail \epsilon} \tag{5}$$

Note the following two properties of the relations: (i) they produce outputs of all possible Nielsen transformation steps applicable with the first symbols on the two sides of the equations and (ii) they include the *trimming* operation.

We compose the introduced relations into a single one, denoted as $\mathcal{T}_{step}$ and defined as $\mathcal{T}_{step} = \mathcal{T}_{x \rightarrowtail \alpha x} \cup \mathcal{T}_{x \rightarrowtail \epsilon}$. The relation $\mathcal{T}_{step}$ can then be used to compute *all successors* of a set of word terms of equations in one step. For a set of word terms S we can compute the $\mathcal{T}_{step}$-image of S to obtain all successors of pairs of word terms in S. The initial configuration, given a word equation $eq{:}\, \mathrm{t}_\ell = \mathrm{t}_r$, is the set $E_{eq} = \{(\mathrm{t}_\ell, \mathrm{t}_r)\}$.

Example 4. Lifting our running example to the introduced notions over sets, we start with the set $E_{eq} = S_0 = \{e_1 = (xay, yx)\}$. After applying $\mathcal{T}_{step}$ on E_{eq}, we obtain the set $S_1 = \{e'_2 = (xay, yx), e'_3 = (ay, y), (axy, yx), (a, \epsilon)\}$. The pairs e'_2 and e'_3 were described earlier, the pair (axy, yx) is obtained by the transformation $\tau_{y \rightarrowtail xy}$, and the pair (a, ϵ) is obtained by the transformation $\tau_{y \rightarrowtail \epsilon}$. If we continue by computing $\mathcal{T}_{step}(S_1)$, we obtain the set $S_2 = S_1 \cup \{(ax, x)\}$, as shown in Fig. 7 (the pair (ax, x) was obtained from (axy, yx) using the transformation $\tau_{y \rightarrowtail \epsilon}$). □

Using the symbolic representation, we can formulate the problem of checking satisfiability of a word equation eq as the task of

- either testing whether $(\epsilon, \epsilon) \in \mathcal{T}^{rt}_{step}(E_{eq})$; if the membership holds, it means that the constraint eq is satisfiable, or
- finding a set (called *unsat-invariant*) E_{inv} such that $E_{eq} \subseteq E_{inv}$, $(\epsilon, \epsilon) \notin E_{inv}$, and $\mathcal{T}_{step}(E_{inv}) \subseteq E_{inv}$, implying that eq is unsatisfiable.

In the following sections, we show how to encode the problem into the RMC framework.

Example 5. To proceed in our running example, when we apply $\mathcal{T}_{step}$ on S_2, we get $\mathcal{T}_{step}(S_2) \subseteq S_2$. Since $e_1 \in S_2$ and $(\epsilon, \epsilon) \notin S_2$, the set S_2 is our unsat-invariant, which means that eq_1 is unsatisfiable. □

3.3. Towards symbolic encoding

Let us now discuss some possible encodings of the word equations satisfiability problem into RMC. Recall that our task is to find an encoding such that the encoded equation (corresponding to initial configurations in RMC) and satisfiability condition (corresponding to destination configurations) are regular languages and the transformation (transition) relation is a rational relation. We start by describing two possible methods of encodings that do not work, analyze why they cannot be used, and then describe a working encoding that we do use.

The first idea about how to encode a set of word equations as a regular language is to encode a pair $e_{eq} = (\mathrm{t}_\ell, \mathrm{t}_r)$ as a word $\mathrm{t}_\ell \cdot ⊜ \cdot \mathrm{t}_r$, where $⊜ \notin \Sigma_{\mathbb{X}}$. One immediately finds out that although the transformations $\tau_{x \rightarrowtail \alpha x}$ and $\tau_{x \rightarrowtail \epsilon}$ are rational (i.e., expressible using a transducer), the transformation τ_{trim}, which removes the longest matching prefix from both sides, is not (a transducer with an unbounded memory to remember the prefix would be required).

The second attempt of an encoding might be to encode $e_{eq} = (\mathrm{t}_\ell, \mathrm{t}_r)$ as a rational binary relation, represented, e.g., by a (not necessarily length-preserving) 2-tape transducer (with one tape for t_ℓ and the other tape for t_r) and use four-tape transducers to represent the transformations (with two tapes for t_ℓ and t_r and two tapes for t'_ℓ and t'_r). The transducers implementing $\tau_{x \rightarrowtail yx}$ and $\tau_{x \rightarrowtail \epsilon}$ can be constructed easily and so can be the transducer implementing τ_{trim}, so this solution looks appealing. One, however, quickly realizes that there is an issue in computing $\mathcal{T}_{step}(E_{eq})$. In particular, since E_{eq} and $\mathcal{T}_{step}$ are both represented as rational relations, the intersection $(E_{eq} \times \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}) \cap \mathcal{T}_{step}$, which needs to be computed first, may not be rational any more. Why? Suppose $E_{eq} = \{\, (a^m b^n, c^m) \mid m, n \geq 0 \,\}$ and $\mathcal{T}_{step} = \{\, (a^m b^n, c^n, \epsilon, \epsilon) \mid m, n \geq 0 \,\}$. Then the intersection $(E_{eq} \times \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}) \cap \mathcal{T}_{step} = \{\, (a^n b^n, c^n, \epsilon, \epsilon) \mid n \geq 0 \,\}$ is clearly not rational any more.

3.4. Symbolic encoding of quadratic equations into RMC

We therefore converge on the following method of representing word equations by a regular language. A set of pairs of word terms is represented as a regular language over a 2-track alphabet with padding $\Sigma^2_{\mathbb{X},\square}$, where $\Sigma_{\mathbb{X},\square} = \Sigma_{\mathbb{X}} \cup \{\square\}$, using an FA. For instance, $e_1 = (xay, yx)$ would be represented by the regular language $\begin{bmatrix} x\,a\,y \\ y\,x\,\square \end{bmatrix} \begin{bmatrix} \square \\ \square \end{bmatrix}^*$. In other words, the equation e_1 has many encodings that differ by the padding, with $\begin{bmatrix} x\,a\,y \\ y\,x\,\square \end{bmatrix}$ being the shortest encoding. The valid representation of the equation contains all of these encodings. On the other hand, Nielsen transformations are represented by (in general, length non-preserving) *binary* rational relations over the 2-track alphabet $\Sigma^2_{\mathbb{X},\square}$ (the first item of each pair refers to an encoding of an equation and the second one refers to the particular transformation applied to the encoding). For instance, the transformation $\tau_{x\rightarrowtail\epsilon}$ is represented by a rational relation containing, e.g., the pair $\left(\begin{bmatrix} x\,a\,y\,\square \\ y\,x\,\square\,\square \end{bmatrix}, \begin{bmatrix} a\,y\,\square \\ y\,\square\,\square \end{bmatrix}\right)$ and the transformation $\tau_{y\rightarrowtail xy}$ is represented by a rational relation containing, e.g., the pair $\left(\begin{bmatrix} x\,a\,y\,\square \\ y\,x\,\square\,\square \end{bmatrix}, \begin{bmatrix} a\,x\,y \\ y\,x\,\square \end{bmatrix}\right)$.

Formally, we first define the *equation encoding function* $\mathsf{eqencode}\colon (\Sigma^*_{\mathbb{X}})^2 \to (\Sigma^2_{\mathbb{X},\square} \setminus \{\begin{bmatrix} \square \\ \square \end{bmatrix}\})^*$ such that for a pair of word terms $\mathsf{t}_\ell = a_1 \dots a_n$ and $\mathsf{t}_r = b_1 \dots b_m$ (without loss of generality we assume that $n \geq m$), we have $\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r) = \begin{bmatrix} a_1 & \cdots & a_m & a_{m+1} & \cdots & a_n \\ b_1 & \cdots & b_m & \square & \cdots & \square \end{bmatrix}$. We also lift eqencode to sets of pairs of word terms $S \subseteq \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}$ as $\mathsf{eqencode}(S) = \{\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r) \mid (\mathsf{t}_\ell, \mathsf{t}_r) \in S\}$.

Let σ be a symbol. We define the *padding* of a word w with respect to σ as the language $\mathsf{pad}_\sigma = \{(w, w') \mid w' \in \{w\}.\{\sigma\}^*\}$, i.e., it is a set of words obtained from w by extending it by an arbitrary number of σ's. Moreover, we also create a (length non-preserving) transducer T_{trim} that performs trimming of its input; this is easy to implement by a two-state transducer that replaces a prefix of symbols of the form $\begin{bmatrix} \beta \\ \beta \end{bmatrix}$ with ϵ, for $\beta \in \Sigma_{\mathbb{X},\square}$. We define the function encode, used for encoding word equations into regular languages, as $\mathsf{encode} = T_{trim} \circ \mathsf{pad}_{\begin{bmatrix} \square \\ \square \end{bmatrix}} \circ \mathsf{eqencode}$, i.e., it takes an encoding of the equation, adds padding, and trims the maximum shared prefix of the two sides of the equation. For example, $\mathsf{encode}(bxay, byx) = \begin{bmatrix} x\,a\,y \\ y\,x\,\square \end{bmatrix}\begin{bmatrix} \square \\ \square \end{bmatrix}^*$. Moreover, for an equation $e \in \Sigma^*_{\mathbb{X}} \times \Sigma^*_{\mathbb{X}}$, a word $w \in \mathsf{encode}(e)$, and a Nielsen rule ρ, we use $w[\rho]$ to denote the set $\mathsf{encode}(\tau_\rho(e))$.

Lemma 4. *Given a word equation $eq\colon \mathsf{t}_\ell = \mathsf{t}_r$ for $\mathsf{t}_\ell, \mathsf{t}_\ell \in \Sigma^*_{\mathbb{X}}$, the set $\mathsf{encode}(eq)$ is regular.*

Proof.

Without loss of generality we assume that $|\mathsf{t}_r| \leq |\mathsf{t}_\ell|$. We give the following MSO(Str) formula that encodes eq:

$$\begin{aligned} \varphi_{eq}(w, w') \triangleq & \bigwedge_{1 \leq k \leq |\mathsf{t}_\ell|} w[k] = \mathsf{t}_\ell[k] \;\wedge \bigwedge_{1 \leq k \leq |\mathsf{t}_r|} w'[k] = \mathsf{t}_r[k] \\ & \wedge \bigwedge_{|\mathsf{t}_r| < k \leq |\mathsf{t}_\ell|} w'[k] = \square \;\wedge \\ & \forall^{\mathbb{P}} p((p > |\mathsf{t}_\ell|) \rightarrow (w[p] = \square \wedge w'[p] = \square)) \end{aligned} \tag{6}$$

From Proposition 2, it follows that $\mathcal{L}(\varphi_{eq})$ is a regular binary relation and, moreover, it can be interpreted as a regular language over the composed alphabet $\Sigma^2_{\mathbb{X},\square}$. Since the image of a regular language with respect to a rational relation (realizing T_{trim}) is also regular (cf. Proposition 1), it follows that $\mathsf{encode}(\mathcal{L}(\varphi_{eq}))$ is also regular. $\square$

Using the presented encoding, when trying to express the $\tau_{x\rightarrowtail\alpha x}$ and $\tau_{x\rightarrowtail\epsilon}$ transformations, we, however, encounter an issue with the need of an unbounded memory. For instance, for the language $L = \begin{bmatrix} x \\ y \end{bmatrix}^*$, the transducer implementing $\tau_{x\rightarrowtail yx}$ would need to remember how many times it has seen x on the first track of its input (indeed, the image of L with respect to $\tau_{x\rightarrowtail yx}$, i.e., the set $\{\mathsf{encode}(u, v) \mid \exists n \geq 0\colon u = (yx)^n \wedge v = y^n\square^n\}$, is no longer regular).

We address this issue in several steps: first, we give a rational relation that correctly represents the transformation rules for cases where the equation eq is quadratic, and extend our algorithm to equations with more occurrences of variables in Section 4. Let us define the following, more general, restriction of $\tau_{x\rightarrowtail\alpha x}$ to equations with at most $i \in \mathbb{N}$ occurrences of variable x:

$$\tau^{\leq i}_{x\rightarrowtail\alpha x} = \tau_{x\rightarrowtail\alpha x} \cap \{((\mathsf{t}_\ell, \mathsf{t}_r), (w, w')) \mid w, w' \in \Sigma^*_{\mathbb{X}}, |\mathsf{t}_\ell.\mathsf{t}_r|_x \leq i\}. \tag{7}$$

We define $\tau^{\leq i}_{x\rightarrowtail\epsilon}$, $\tau^{\leq i}_{x\mapsto\alpha x}$, and $\tau^{\leq i}_{x\mapsto\epsilon}$ similarly.

3.4.1. Encoding Nielsen transformations as rational relations

Next, in order to be able to perform the operations given by $\tau^{\leq i}_{x\rightarrowtail\epsilon}$ and $\tau^{\leq i}_{x\rightarrowtail\alpha x}$ on our encoding within the RMC framework, we need to encode them as rational relations. In this section, we define rational relations $\mathcal{T}^{\leq i}_{x\rightarrowtail\epsilon}$ and $\mathcal{T}^{\leq i}_{x\rightarrowtail\alpha x}$ that do exactly this encoding. We obtain these relations in successive steps, by defining several intermediate formulae with the transformation as subscript and the number of variables as superscript, e.g., $\varphi^{\leq n}_{x\rightarrowtail\epsilon}$ and $\psi^{n}_{x\rightarrowtail\alpha x}$.

We begin with defining some useful MSO(Str) predicates for an MSO(Str) string variable w, a word constraint variable x, and positions $k_1, \dots, k_m$.

$$\mathit{ordered}(k_1, \dots, k_m) \triangleq \bigwedge_{1 \leq i < m} k_i < k_{i+1} \tag{8}$$

$$\mathit{alleq}^w_x(k_1, \dots, k_m) \triangleq \bigwedge_{1 \leq i \leq m} w[k_i] = x \tag{9}$$

$$\begin{aligned} \mathit{occur}^w_x(k_1, \dots, k_m) \triangleq\; & \mathit{ordered}(k_1, \dots, k_m) \wedge \mathit{alleq}^w_x(k_1, \dots, k_m) \\ & \wedge \forall^{\mathbb{P}} j\Big(w[j] = x \rightarrow \textstyle\bigvee_{1 \leq i \leq m} j = k_i\Big) \end{aligned} \tag{10}$$

We use the following MSO(Str) formula to define the transformation $x \mapsto \epsilon$ for n occurrences of x in a single string. The formula guesses n positions of x and then ensures that all symbols in w' are correctly shifted. In particular, the symbols on positions smaller than i_1 are copied from w without change. Symbols in w on positions between i_ℓ and $i_{\ell+1}$ are shifted ℓ positions to the left in w'. And the remaining positions in w' are filled with $\square$'s.

$$\begin{aligned} \psi^n_{x\mapsto\epsilon}(w, w') \triangleq \exists^{\mathbb{P}} i_1, \dots, i_n \Big(\; & \mathit{occur}^w_x(i_1, \dots, i_n) \wedge \\ & \forall^{\mathbb{P}} j(j < i_1 \rightarrow w'[j] = w[j]) \wedge \\ & \bigwedge_{1 \leq k < n} \forall^{\mathbb{P}} j((i_k < j < i_{k+1}) \rightarrow w'[j-k] = w[j]) \wedge \\ & \forall^{\mathbb{P}} j(i_n < j \rightarrow w'[j-n] = w[j]) \wedge \\ & \bigwedge_{1 \leq k \leq n} w'[\$ - k] = \square \Big), \end{aligned} \tag{11}$$

where $w'[\$ - k] = \square$ stands for the formula $\exists^{\mathbb{P}} r \exists^{\mathbb{P}} s(r = \$ \wedge r = s + k \wedge w'[s] = \square)$ (cf. Fig. 4). We extend $\psi^n_{x\mapsto\epsilon}$ to describe the relation on pairs of strings:

$$\psi'^{n}_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}_r, \mathsf{t}'_\ell, \mathsf{t}'_r) \triangleq \bigvee_{0 \leq k \leq n} \psi^k_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}'_\ell) \wedge \psi^{n-k}_{x\mapsto\epsilon}(\mathsf{t}_r, \mathsf{t}'_r) \tag{12}$$

$$\psi'^{\leq n}_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}_r, \mathsf{t}'_\ell, \mathsf{t}'_r) \triangleq \bigvee_{0 \leq k \leq n} \psi'^{k}_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}_r, \mathsf{t}'_\ell, \mathsf{t}'_r) \tag{13}$$

$$\varphi^{\leq n}_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}_r, \mathsf{t}'_\ell, \mathsf{t}'_r) \triangleq (\mathsf{t}_\ell[1] = x \vee \mathsf{t}_r[1] = x) \wedge \psi'^{\leq n}_{x\mapsto\epsilon}(\mathsf{t}_\ell, \mathsf{t}_r, \mathsf{t}'_\ell, \mathsf{t}'_r) \tag{14}$$

Next, we define the transformation $x \mapsto \alpha x$ for n occurrences of x in a single string. The formula guesses n positions of x in w. Then, it ensures that on $i_\ell + \ell$ position in w' there is the symbol α and all other symbols from w are copied to the correct positions in

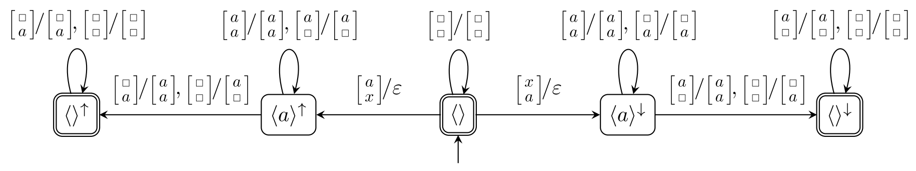

$\begin{bmatrix}\square\\a\end{bmatrix}/\begin{bmatrix}\square\\a\end{bmatrix}, \begin{bmatrix}\square\\\square\end{bmatrix}/\begin{bmatrix}\square\\\square\end{bmatrix}$
$\begin{bmatrix}a\\a\end{bmatrix}/\begin{bmatrix}a\\a\end{bmatrix}, \begin{bmatrix}a\\\square\end{bmatrix}/\begin{bmatrix}a\\\square\end{bmatrix}$
$\begin{bmatrix}\square\\\square\end{bmatrix}/\begin{bmatrix}\square\\\square\end{bmatrix}$
$\begin{bmatrix}a\\a\end{bmatrix}/\begin{bmatrix}a\\a\end{bmatrix}, \begin{bmatrix}\square\\a\end{bmatrix}/\begin{bmatrix}\square\\a\end{bmatrix}$
$\begin{bmatrix}a\\\square\end{bmatrix}/\begin{bmatrix}a\\\square\end{bmatrix}, \begin{bmatrix}\square\\\square\end{bmatrix}/\begin{bmatrix}\square\\\square\end{bmatrix}$
$\begin{bmatrix}\square\\a\end{bmatrix}/\begin{bmatrix}a\\a\end{bmatrix}, \begin{bmatrix}\square\\\square\end{bmatrix}/\begin{bmatrix}a\\\square\end{bmatrix}$
$\begin{bmatrix}a\\x\end{bmatrix}/\varepsilon$
$\begin{bmatrix}x\\a\end{bmatrix}/\varepsilon$
$\begin{bmatrix}a\\\square\end{bmatrix}/\begin{bmatrix}a\\a\end{bmatrix}, \begin{bmatrix}\square\\\square\end{bmatrix}/\begin{bmatrix}\square\\a\end{bmatrix}$
$\langle\rangle^{\uparrow}$
$\langle a\rangle^{\uparrow}$
$\langle\rangle$
$\langle a\rangle^{\downarrow}$
$\langle\rangle^{\downarrow}$

Fig. 8. Example of a transducer realizing the relation $T^{\leq 1}_{x\mapsto\epsilon}$ for the set of variables $\mathbb{X} = \{x\}$ and the alphabet $\Sigma = \{a\}$. The names of the states of the transducers denote symbols that the transducers remember to output on the tape given by the symbols $\uparrow$ and $\downarrow$.

w'. In particular, symbols in w on positions between i_ℓ and $i_{\ell+1}$ are shifted ℓ positions to the right in w'.

$$\begin{aligned}\psi^n_{x\mapsto\alpha x}(w, w') \triangleq \exists^{\mathbb{P}} i_1, \ldots, i_n \Big(& occur^w_x(i_1, \ldots, i_n) \wedge \\ & \forall^{\mathbb{P}} j(j \leq i_1 \rightarrow w'[j] = w[j]) \wedge \\ & \bigwedge_{1\leq k<n} w'[i_k + k] = \alpha \wedge \\ & \quad \forall^{\mathbb{P}} j((i_k < j \leq i_{k+1}) \rightarrow w'[j+k] = w[j]) \wedge \\ & w'[i_n + n] = \alpha \wedge \\ & \forall^{\mathbb{P}} j(i_n < j \rightarrow w'[j+n] = w[j]) \wedge \\ & \bigwedge_{1\leq k\leq n} w[\$ - k] = \square \Big)\end{aligned} \tag{15}$$

We extend $\psi^n_{x\mapsto\alpha x}$ to describe the relation on pairs of strings:

$$\psi'^{n}_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}_r, \mathrm{t}'_\ell, \mathrm{t}'_r) \triangleq \bigvee_{0\leq k\leq n} \psi^k_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}'_\ell) \wedge \psi^{n-k}_{x\mapsto\alpha x}(\mathrm{t}_r, \mathrm{t}'_r) \tag{16}$$

$$\psi'^{\leq n}_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}_r, \mathrm{t}'_\ell, \mathrm{t}'_r) \triangleq \bigvee_{0\leq k\leq n} \psi'^{k}_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}_r, \mathrm{t}'_\ell, \mathrm{t}'_r) \tag{17}$$

$$\begin{aligned}\varphi^{\leq n}_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}_r, \mathrm{t}'_\ell, \mathrm{t}'_r) \triangleq & \big((\mathrm{t}_\ell[1] = x \wedge \mathrm{t}_r[1] = \alpha) \vee (\mathrm{t}_\ell[1] = \alpha \\ & \wedge \mathrm{t}_r[1] = x) \big) \\ & \wedge \psi'^{\leq n}_{x\mapsto\alpha x}(\mathrm{t}_\ell, \mathrm{t}_r, \mathrm{t}'_\ell, \mathrm{t}'_r)\end{aligned} \tag{18}$$

The constructed formulae $\varphi^{\leq n}_{x\mapsto\epsilon}$ and $\varphi^{\leq n}_{x\mapsto\alpha x}$ describe regular relations with arity 4 over the alphabet $\Sigma_{\mathbb{X},\square}$. Since they are regular (i.e., length-preserving), they can also be interpreted as binary relations over the composed alphabet $\Sigma^2_{\mathbb{X},\square}$. This interpretation can easily be done by modifying a length-preserving transducer corresponding to $\varphi^{\leq n}_{x\mapsto\epsilon}$ and $\varphi^{\leq n}_{x\mapsto\alpha x}$ respectively in a way that each transition $q\text{-}\{a_1,a_2,a_3,a_4\}\!\!\rightarrow r$ is replaced by the transition $q\text{-}\{\begin{bmatrix}a_1\\a_2\end{bmatrix},\begin{bmatrix}a_3\\a_4\end{bmatrix}\}\!\!\rightarrow r$ (with the same sets of states). When interpreted as binary relations in this way, they denote the relations containing pairs (without loss of generality, we show this only for $\varphi^{\leq n}_{x\mapsto\epsilon}$)

$$\left(\begin{bmatrix}u_1 & \cdots & u_m & u_{m+1} & \cdots & u_n\\ v_1 & \cdots & v_m & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^i, \begin{bmatrix}w_1 & \cdots & w_k & w_{k+1} & \cdots & w_\ell\\ z_1 & \cdots & z_k & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^j\right)$$

such that

- $\tau^{\leq n}_{x\mapsto\epsilon}((u_1 \ldots u_n, v_1 \ldots v_m)) = (w_1 \ldots w_\ell, z_1 \ldots z_k)$ for $\varphi^{\leq n}_{x\mapsto\epsilon}$,
- $\tau^{\leq n}_{x\mapsto\alpha x}((u_1 \ldots u_n, v_1 \ldots v_m)) = (w_1 \ldots w_\ell, z_1 \ldots z_k)$ for $\varphi^{\leq n}_{x\mapsto\alpha x}$, and
- $\max(m, n) + i = \max(k, \ell) + j$.

Let us now consider $\varphi^{\leq n}_{x\mapsto\epsilon}$. We note that for every $((u, v), (w, z)) \in \tau^{\leq n}_{x\mapsto\epsilon}$, there will indeed be a corresponding pair in $\varphi^{\leq n}_{x\mapsto\epsilon}$ (actually, there will be infinitely many such pairs that differ in the number of used padding symbols).

In order to get closer to $\tau_{x\rightarrowtail\epsilon}$, we need to modify the relation of $\varphi^{\leq n}_{x\mapsto\epsilon}$ to also perform trimming of the shared prefix. We do this modification by taking the (length-preserving) two-track transducer $T^{\leq n}_{x\mapsto\epsilon}$ that recognizes $\varphi^{\leq n}_{x\mapsto\epsilon}$ (it can be constructed due to Proposition 2). Moreover, we also create a (length non-preserving) transducer T_{trim} that performs trimming of its input; this is easy to implement by a two-state transducer that replaces a prefix of symbols of the form $\begin{bmatrix}\beta\\\beta\end{bmatrix}$ with ϵ, for $\beta \in \Sigma_{\mathbb{X},\square}$. By composing the two transducers, we obtain $T^{\leq n}_{x\rightarrowtail\epsilon} = T_{trim} \circ T^{\leq n}_{x\mapsto\epsilon}$. An example of a transducer realizing $T^{\leq 1}_{x\mapsto\epsilon}$ is shown in Fig. 8.

We can repeat the previous reasoning for $\varphi^{\leq n}_{x\mapsto\alpha x}$ in a similar way to obtain the (length non-preserving) transducer $T^{\leq n}_{x\rightarrowtail\alpha x}$.

Algorithm 1: Solving a string constraint φ using RMC

```
  Input: Encoding $\mathcal{I}$ of a formula $\varphi$ (the initial set),
         transformers $\mathcal{T}_{x\rightarrowtail\alpha x}$, $\mathcal{T}_{x\rightarrowtail\epsilon}$, and
         the destination set $\mathcal{D}$
  Output: A model of $\varphi$ if $\varphi$ is satisfiable, false otherwise
1 $reach_0 := \mathcal{I}$;
2 $processed := \emptyset$;
3 $\mathcal{T} := \mathcal{T}_{x\rightarrowtail\alpha x} \cup \mathcal{T}_{x\rightarrowtail\epsilon}$;
4 $i := 0$;
5 while $reach_i \not\subseteq processed$ do
6     if $\mathcal{D} \cap reach_i \neq \emptyset$ then
7         return
            $ExtractModel(\{\mathcal{T}\}_{j\in\{0,\ldots,i\}}, \mathcal{D}, reach_0, \ldots, reach_i)$;
8     $processed := processed \cup reach_i$;
9     $reach_{i+1} := satur \circ \mathcal{T}(reach_i)$;        // $satur^\bullet \circ \mathcal{T}(reach_i)$
10    $i$++;
11 return false;
```

Lemma 5. *It holds that* $\tau^{\leq n}_{x\rightarrowtail\epsilon}((u_1 \ldots u_n, v_1 \ldots v_m)) = (w_1 \ldots w_\ell, z_1 \ldots z_k)$ *iff*

$$\left(\begin{bmatrix}u_1 & \cdots & u_m & u_{m+1} & \cdots & u_n\\ v_1 & \cdots & v_m & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^i, \begin{bmatrix}w_1 & \cdots & w_k & w_{k+1} & \cdots & w_\ell\\ z_1 & \cdots & z_k & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^j\right) \in \mathcal{L}(T^{\leq n}_{x\rightarrowtail\epsilon})$$

for some $i, j \in \mathbb{N}$.

Further, it holds that $\tau^{\leq n}_{x\rightarrowtail\alpha x}((u_1 \ldots u_n, v_1 \ldots v_m)) = (w_1 \ldots w_\ell, z_1 \ldots z_k)$ *iff*

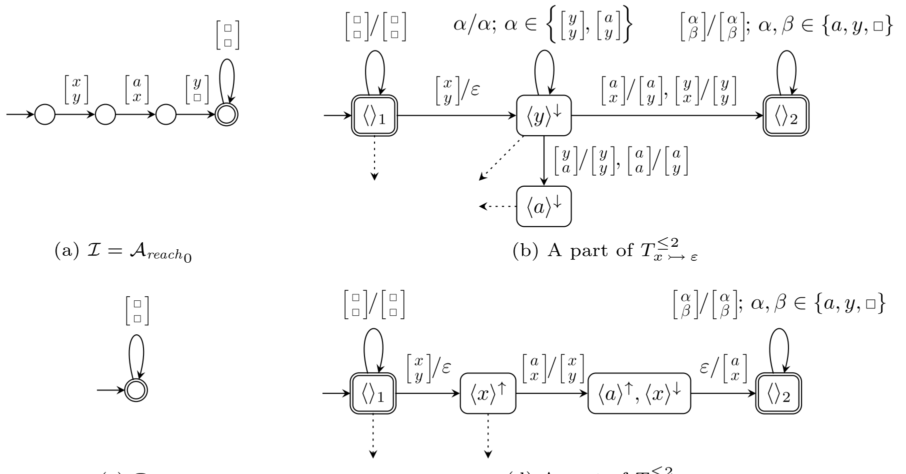
$$\left(\begin{bmatrix}u_1 & \cdots & u_m & u_{m+1} & \cdots & u_n\\ v_1 & \cdots & v_m & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^i, \begin{bmatrix}w_1 & \cdots & w_k & w_{k+1} & \cdots & w_\ell\\ z_1 & \cdots & z_k & \square & \cdots & \square\end{bmatrix}\begin{bmatrix}\square\\\square\end{bmatrix}^j\right) \in \mathcal{L}(T^{\leq n}_{x\rightarrowtail\alpha x})$$

for some $i, j \in \mathbb{N}$.

Proof. The proof follows from the above described construction of transducers $T^{\leq n}_{x\rightarrowtail\alpha x}$ and $T^{\leq n}_{x\rightarrowtail\epsilon}$. □

3.4.2. RMC for quadratic equations

In Algorithm 1, we give a high-level algorithm for solving string constraints using RMC. The algorithm is parameterized by the following: (i) a regular language $\mathcal{I}$ encoding a formula φ (the initial set), (ii) rational relations given by the transducers $\mathcal{T}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}_{x\rightarrowtail\epsilon}$, and (iii) the destination set $\mathcal{D}$ (also given as a regular language). The algorithm tries to solve the RMC problem $(\mathcal{I}, \mathcal{T}_{x\rightarrowtail\alpha x} \cup \mathcal{T}_{x\rightarrowtail\epsilon}, \mathcal{D})$ by an iterative unfolding of the



$\langle\rangle_1$
$\langle y\rangle^{\downarrow}$
$\langle\rangle_2$
$\langle a\rangle^{\downarrow}$
$\langle x\rangle^{\uparrow}$
$\langle a\rangle^{\uparrow}, \langle x\rangle^{\downarrow}$
$\alpha/\alpha;\ \alpha \in \left\{\left[\begin{smallmatrix}y\\y\end{smallmatrix}\right], \left[\begin{smallmatrix}a\\y\end{smallmatrix}\right]\right\}$
$\left[\begin{smallmatrix}\alpha\\\beta\end{smallmatrix}\right]/\left[\begin{smallmatrix}\alpha\\\beta\end{smallmatrix}\right];\ \alpha,\beta \in \{a, y, \square\}$

(a) $\mathcal{I} = \mathcal{A}_{reach_0}$ (b) A part of $T^{\le 2}_{x \rightarrowtail \varepsilon}$

(c) $\mathcal{D}$ (d) A part of $T^{\le 2}_{x \rightarrowtail yx}$

Fig. 9. Input of Algorithm 1 for solving the word equation $xay = yx$. The size of $\mathcal{T}^{eq}_{x\rightarrowtail\epsilon}$ and $\mathcal{T}^{eq}_{x\rightarrowtail yx}$ would be prohibitively large, so we only give relevant parts of transducers $T^{\le 2}_{x\rightarrowtail\epsilon}$ and $T^{\le 2}_{x\rightarrowtail yx}$. Some transitions use shorthand notation with the obvious meaning.

Function ExtractModel($\{\mathcal{T}_j\}_{j\in\{0,\dots,i\}}, \mathcal{D}, reach_0, \dots, reach_i$)

```
1 $I(a) := a$ for each $a \in \Sigma$, $I(x) := \epsilon$ for each $x \in \mathbb{X}$;
2 let $w_i \in \mathcal{D} \cap reach_i$;
3 for $\ell = i$ downto 1 do
4     let $w_{\ell-1} \in \mathcal{T}_\ell^{-1}(w_\ell) \cap reach_{\ell-1}$;
5     let $\rho$ be a rule s.t. $w_\ell \in w_{\ell-1}[\rho]$;
6     if $\rho = y \rightarrowtail \alpha y$ then
7         $I(y) := I(\alpha).I(y)$;
8 return $I$;
```

transition relation $\mathcal{T}$ computed in Line 3, looking for an element w_i from $\mathcal{D}$. If such an element is found in the set $reach_i$, we call Function ExtractModel to extract a model of the original word equation by starting a backward run from w_i, computing pre-images $w_{i-1}, \dots, w_1$ over transformers $\mathcal{T}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}_{x\rightarrowtail\epsilon}$ (restricting them to $reach_j$ for every w_j), while updating values of the variables according to the transformations that were performed. Note that *ExtractModel* uses a more general interface allowing to specify a transducer for each backward step (Line 4 of Function ExtractModel). This is utilized later in Section 4; here, we just pass i copies of $\mathcal{T}$. Algorithm 1 also employs *saturation* of the sets of reachable configurations defined as:

$$satur(L) = \left\{u \;\middle|\; w \in L,\ w \in u.\left[\begin{smallmatrix}\square\\\square\end{smallmatrix}\right]^*\right\}. \tag{19}$$

Intuitively, $satur(L)$ removes some occurrences (possibly none of them) of the padding symbol at the end of all words from L. If L is a regular language, $satur(L)$ is regular as well (from an FA representing L we can get $satur(L)$ by saturating its transitions over the padding symbol). We saturate the sets of reachable configurations, because we want to keep the shortest words (i.e., words without padding symbols)—e.g., the transformer $\mathcal{T}^{eq}_{x\rightarrowtail\epsilon}$ need not generate all shortest words.

Algorithm 1 follows a *breadth-first search* (BFS) strategy: from the initial set $\mathcal{I}$, we apply both transformers $\mathcal{T}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}_{x\rightarrowtail\epsilon}$ on all elements of $\mathcal{I}$ at the same time, before repeatedly applying the transformers on the result. This corresponds to a breadth-first application of the transformers if we applied them one element of $\mathcal{I}$ at a time.

Our first instantiation of the algorithm is for checking satisfiability of a single quadratic word equation $eq\colon \mathsf{t}_\ell = \mathsf{t}_r$. We instantiate the RMC problem as the tuple $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x\rightarrowtail\alpha x} \cup \mathcal{T}^{eq}_{x\rightarrowtail\epsilon}, \mathcal{D}^{eq})$ where

$$\mathcal{I}^{eq} = \mathsf{encode}(\mathsf{t}_\ell, \mathsf{t}_r) \qquad \mathcal{T}^{eq}_{x\rightarrowtail\alpha x} = \bigcup_{y\in\mathbb{X},\alpha\in\Sigma_{\mathbb{X}}} T^{\le 2}_{y\rightarrowtail\alpha y} \qquad \mathcal{D}^{eq} = \left\{\left[\begin{smallmatrix}\square\\\square\end{smallmatrix}\right]\right\}^*$$

$$\mathcal{T}^{eq}_{x\rightarrowtail\epsilon} = \bigcup_{y\in\mathbb{X}} T^{\le 2}_{y\rightarrowtail\epsilon}$$

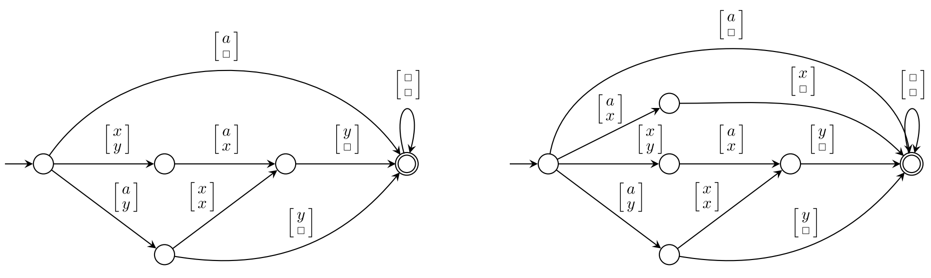
Lemma 6. *Algorithm 1 instantiated with $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x\rightarrowtail\alpha x} \cup \mathcal{T}^{eq}_{x\rightarrowtail\epsilon}, \mathcal{D}^{eq})$ is sound, complete, and terminating if $eq\colon \mathsf{t}_\ell = \mathsf{t}_r$ is quadratic.*

Proof. We encode the nodes of a Nielsen proof graph as strings. The initial node $\mathsf{t}_\ell = \mathsf{t}_r$ corresponds to a string from $\mathcal{I}^{eq}$. Due to the padding, the final node $\epsilon = \epsilon$ corresponds to a string from $\mathcal{D}^{eq}$. The relations $\mathcal{T}^{eq}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}^{eq}_{x\rightarrowtail\epsilon}$ implement Nielsen rules $x \rightarrowtail \alpha x$ and $x \rightarrowtail \epsilon$. From Lemma 3 we have that the Nielsen proof graph is finite. Since our approach implements the *breadth-first search* (BFS) strategy, it is sound, complete, and terminating (see Fig. 10). □

Example 6. Consider the word equation $eq_1\colon xay = yx$ from Section 3.1. We apply Algorithm 1 on the encoded equation $\mathcal{I} = \mathsf{encode}(xay, yx) = \left[\begin{smallmatrix}x\,a\,y\\y\,x\,\square\end{smallmatrix}\right]\left[\begin{smallmatrix}\square\\\square\end{smallmatrix}\right]^*$. The inputs of the algorithm are in Fig. 9. In the first iteration of the main loop, the regular set *processed*, represented by its minimal FA $\mathcal{A}_{reach_1}$, is given in Fig. 10(a) (the FA also corresponds to $reach_1$ and represents the set S_1 from Example 4). In particular, consider, e.g.,

$$\left(\left[\begin{smallmatrix}x\,a\,y\\y\,x\,\square\end{smallmatrix}\right], \left[\begin{smallmatrix}a\,y\,\square\\y\,\square\,\square\end{smallmatrix}\right]\right) \in \mathcal{T}^{eq}_{x\rightarrowtail\epsilon}.$$

After saturation we get that $\left[\begin{smallmatrix}a\,y\\y\,\square\end{smallmatrix}\right] \in reach_1$. In the second (and also the last) iteration, the set *processed* is given by the minimal FA $\mathcal{A}_{reach_2}$ in Fig. 10(b) (which also corresponds to $reach_2$ and $reach_3$, and the set S_2 from Example 4). Since $reach_3 \subseteq processed$, the algorithm terminates with false, establishing the unsatisfiability of eq_1. □



(a) $\mathcal{A}_{reach_1}$ (b) $\mathcal{A}_{reach_2}$

Fig. 10. Examples of finite automata $\mathcal{A}_{reach_1}$, $\mathcal{A}_{reach_2}$ representing sets of configurations reachable in one and two steps, respectively, when solving the word equation $xay = yx$ using Algorithm 1.

4. Solving a system of word equations using RMC

In the previous section we described how to solve a single quadratic word equation in the RMC framework. In this section we focus on an extension of this approach to handle a system of word equations of the form

$$\Phi : \mathsf{t}_\ell^1 = \mathsf{t}_r^1 \wedge \mathsf{t}_\ell^2 = \mathsf{t}_r^2 \wedge \cdots \wedge \mathsf{t}_\ell^n = \mathsf{t}_r^n. \tag{20}$$

In the first step we need to encode the system Φ as a regular language. For this we extend the encode function to a system of word equations by defining

$$\mathsf{encode}(\Phi) = \mathsf{encode}(\mathsf{t}_\ell^1, \mathsf{t}_r^1).\left\{\begin{bmatrix}\#\\\#\end{bmatrix}\right\}.\cdots.\left\{\begin{bmatrix}\#\\\#\end{bmatrix}\right\}.\mathsf{encode}(\mathsf{t}_\ell^n, \mathsf{t}_r^n), \tag{21}$$

where # is a delimiter symbol, $\# \notin \Sigma_{\mathbb{X},\square}$. From Lemma 4 we know that $\mathsf{encode}(\mathsf{t}_\ell^i, \mathsf{t}_r^i)$ is regular for all $1 \leq i \leq n$. Moreover, since regular languages are closed under concatenation (Proposition 1), the set $\mathsf{encode}(\Phi)$ is also regular. Because each equation is now separated by a delimiter, we need to extend the destination set to $\left\{\begin{bmatrix}\square\\\square\end{bmatrix}, \begin{bmatrix}\#\\\#\end{bmatrix}\right\}^*$.

For the transition relation, we need to extend $\tau^{\leq i}_{x \rightarrowtail \alpha x}$ and $\tau^{\leq i}_{x \rightarrowtail \epsilon}$ from the previous section to support delimiters. An application of a rule $x \rightarrowtail \alpha x$ on a system of equations can be described as follows: the rule $x \rightarrowtail \alpha x$ is applied to the first non-empty equation and the rest of the equations are modified according to the substitution $x \mapsto \alpha x$. The substitution on the other equations is performed regardless of their first symbols. The procedure is analogous for the rule $x \rightarrowtail \epsilon$. A series of applications of the rules can reduce the number of equations, which then leads to a string in our encoding with a prefix from $\left\{\begin{bmatrix}\square\\\square\end{bmatrix}, \begin{bmatrix}\#\\\#\end{bmatrix}\right\}^*$. The relation implementing $x \rightarrowtail \alpha x$ or $x \rightarrowtail \epsilon$ on an encoded system of equations skips this prefix. Formally, the rule $x \rightarrowtail \alpha x$ for a system of equations where every equation has at most i occurrences of every variable is given by the following relation:

$$T^{\wedge eq,i}_{x \rightarrowtail \alpha x} = T_{skip}.T^{\leq i}_{x \rightarrowtail \alpha x}.\left(\left\{\begin{bmatrix}\#\\\#\end{bmatrix} \mapsto \begin{bmatrix}\#\\\#\end{bmatrix}\right\}.(T_{trim} \circ S^{\leq i}_{x \mapsto \alpha x})\right)^*, \tag{22}$$

where $T_{skip} = \left\{\begin{bmatrix}\square\\\square\end{bmatrix} \mapsto \begin{bmatrix}\square\\\square\end{bmatrix}, \begin{bmatrix}\#\\\#\end{bmatrix} \mapsto \begin{bmatrix}\#\\\#\end{bmatrix}\right\}^*$ and $S^{\leq i}_{x \mapsto \alpha x}$ is the binary transducer representing the formula $\psi'^{\leq i}_{x \mapsto \alpha x}$ from Section 3.4.1 (which does not look at the first symbol on the tape). The relation $T^{\wedge eq,i}_{x \rightarrowtail \epsilon}$ is defined in a similar manner. By construction and from the closure properties of rational relations (cf. Proposition 1), it is clear that $T^{\wedge eq,i}_{x \rightarrowtail \alpha x}$ and $T^{\wedge eq,i}_{x \rightarrowtail \epsilon}$ are rational.

4.1. Quadratic case

When Φ is quadratic, its satisfiability problem can be reduced to an RMC problem $(\mathcal{I}^{\wedge eq}_\Phi, \mathcal{T}^{\wedge eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{\wedge eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{\wedge eq})$ instantiating Algorithm 1 where

$$\mathcal{I}^{\wedge eq}_\Phi = \mathsf{encode}(\Phi) \qquad \mathcal{T}^{\wedge eq}_{x \rightarrowtail \alpha x} = \bigcup_{y \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} T^{\wedge eq,2}_{y \rightarrowtail \alpha y} \qquad \mathcal{D}^{\wedge eq} = \left\{\begin{bmatrix}\square\\\square\end{bmatrix}, \begin{bmatrix}\#\\\#\end{bmatrix}\right\}^*$$

$$\mathcal{T}^{\wedge eq}_{x \rightarrowtail \epsilon} = \bigcup_{y \in \mathbb{X}} T^{\wedge eq,2}_{y \rightarrowtail \epsilon}$$

Rationality of $\mathcal{T}^{\wedge eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{\wedge eq}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 1. In addition, we also need to modify Algorithm 1 such that in Line 9, we substitute *satur* with *satur*$^\bullet$ defined as follows:

$$satur^\bullet(L) = \left\{ u_1.u_2 \ldots u_k \;\middle|\; w \in L, w \in u_1.\begin{bmatrix}\square\\\square\end{bmatrix}^+.u_2.\begin{bmatrix}\square\\\square\end{bmatrix}^+ \cdots u_k \text{ for some } k \in \mathbb{N} \right\}.$$

Intuitively, *satur*$^\bullet(L)$ modifies *satur* to take into account the fact that we now work with several word equations encoded into a single word, where they are separated by delimiters. Now, *satur*$^\bullet$ does not only remove paddings at the end of the word (when $k = 1$), but also in the middle.

The soundness and completeness of our procedure for a system of quadratic word equations is summarized by the following lemma.

Lemma 7. *Algorithm 1 instantiated with* $(\mathcal{I}^{\wedge eq}_\Phi, \mathcal{T}^{\wedge eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{\wedge eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{\wedge eq})$ *is sound, complete, and terminating if* Φ *is quadratic.*

Proof. We encode the nodes of a Nielsen proof graph as strings. The initial node Φ corresponds to a string from $\mathcal{I}^{\wedge eq}_\Phi$ (conjunction can be seen as the delimiter #). Because of the padding, the final node $\epsilon = \epsilon \wedge \cdots \wedge \epsilon = \epsilon$ corresponds to a string from $\mathcal{D}^{\wedge eq}$. The relations $\mathcal{T}^{\wedge eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{\wedge eq}_{x \rightarrowtail \epsilon}$ implement the Nielsen rules $x \rightarrowtail \alpha x$ and $x \rightarrowtail \epsilon$. From Lemma 3 we have that the Nielsen proof graph is finite (and hence the potential final node is in a finite depth). Since our approach implements the BFS strategy, it is both sound, complete, and terminating. □

4.2. General case

Let us now consider the general case when the system Φ is not quadratic. In this section, we show that this general case is also reducible to an extended version of RMC.

We first apply Algorithm 2 to a general system of string constraints Φ in order to get an equisatisfiable cubic system of word equations Φ'. If the input of the transformation is a system of equations with n symbols, then the output of the transformation will, in the worst case, contain $\frac{n}{2}$ additional word equations and $\frac{n}{2}$ additional literals, so the transformation is linear. Then, we

Algorithm 2: Transformation to a cubic system of equations

```
  Input: System of word equations $\Phi$
  Output: Equisatisfiable cubic system of word equations $\Psi$
1 $\Psi := \Phi$;
2 while $\exists x \in \mathbb{X}$ s.t. $x$ occurs more than three times in $\Psi$ do
3     Replace two occurrences of $x$ in $\Phi$ by a fresh word variable $x'$ to obtain a new system $\Psi'$;
4     $\Psi := (\Psi' \wedge x = x')$;
5 return $\Psi$;
```

can use the transition relations $T^{\wedge eq,3}_{x \rightarrowtail \alpha x}$ and $T^{\wedge eq,3}_{x \rightarrowtail \epsilon}$ to construct transformations of the encoded system Φ'.

Lemma 8. *Any system of word equations can be transformed by Algorithm 2 to an equisatisfiable cubic system of word equations.*

Proof. Let Φ be the input system of word equations. Observe that in every iteration of Algorithm 2, the number of occurrences of a variable x is decreased by one and a new variable x' with exactly three occurrences is introduced. □

One more issue we need to solve is to make sure that we work with a cubic system of word equations in every step of our algorithm. It may happen that a transformation of the type $x \rightarrowtail yx$ increases the number of occurrences of the variable y by one, so if there had already been three occurrence of y before the transformation, the result will not be cubic any more. More specifically, assume a cubic system of word equations $x.\mathrm{t}_\ell = y.\mathrm{t}_r \wedge \Phi$, where x and y are distinct word variables and t_ℓ and t_r are word terms. If we apply the transformation $x \rightarrowtail yx$, we will obtain $x(\mathrm{t}_\ell[x \mapsto yx]) = \mathrm{t}_r[x \mapsto yx] \wedge \Phi[x \mapsto yx]$. Observe that (i) the number of occurrences of y is first *reduced by one* because the first y on the right-hand side of $x.\mathrm{t}_\ell = y.\mathrm{t}_r$ is removed and (ii) the number of occurrences of y can be at most *increased by two* because there exist at most two occurrences of x in t_ℓ, t_r, and Φ. Therefore, after the transformation $x \rightarrowtail yx$, a cubic system of word equations might become a *(y-)quartic system of word equations* (at most four occurrences of the variable y and at most three occurrences of any other variable). For this reason, we need to apply the conversion to the cubic system after each transformation.

Given a fresh variable v, we use $\mathcal{C}_v$ to denote the transformation from a single-quartic system of word equations to a cubic system of equations using the fresh variable v.

Lemma 9. *The relation $T_{\mathcal{C}_v}$ performing the transformation $\mathcal{C}_v$ on an encoded single-quartic system of equations is rational.*

Proof. We show how we can create a transducer for the transformation from a single-quartic system of word equations to a cubic system of word equations.

In the first step, we create the transducer $\mathcal{T}^{sq}_{x,x_i}$ that accepts only input that is an encoding of a x-quartic system of word equations. This can be done by using states to trace the number of occurrences of variables (we only need to count up to four). For an encoding of a x-quartic system of word equations, the transducer $\mathcal{T}^{sq}_{x,x_i}$ returns an encoding that is obtained by replacing the first two occurrences of x from the input to x_i and concatenating the language $\left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]\left[\begin{smallmatrix}x\\x_i\end{smallmatrix}\right]\left[\begin{smallmatrix}\square\\\square\end{smallmatrix}\right]^*$ at the end.

In the second step, we create the transducer $\mathcal{T}_{cub}$ that accepts only encodings of a cubic system of word equations and returns the same encodings.

Algorithm 3: Solving a general string constraint φ using RMC

```
   Input: Encoding $\mathcal{I}$ of a formula $\varphi$ (the initial set),
          ordered set of indices $\mathbb{V} = \{v_1, v_2, \ldots\}$,
          parameterized transformers $\mathcal{T}^v_{x \rightarrowtail \alpha x}$, $\mathcal{T}^v_{x \rightarrowtail \epsilon}$ where
          $v \in \mathbb{V}$, and
          the destination set $\mathcal{D}$
   Output: A model of $\varphi$ if $\varphi$ is satisfiable, false otherwise
1  $reach_0 := \mathcal{I}$;
2  $processed := \emptyset$;
3  $\mathcal{T}^v := \mathcal{T}^v_{x \rightarrowtail \alpha x} \cup \mathcal{T}^v_{x \rightarrowtail \epsilon}$;
4  $i := 0$;
5  while $reach_i \not\subseteq processed$ do
6      if $\mathcal{D} \cap reach_i \neq \emptyset$ then
7          return
             $ExtractModel(\{\mathcal{T}^{v_j}\}_{j \in \{0,\ldots,i\}}, \mathcal{D}, reach_0, \ldots, reach_i)$;
8      $processed := processed \cup reach_i$;
9      $reach_{i+1} := satur^{\bullet} \circ \mathcal{T}^{v_i}(reach_i)$;
10     $\mathbb{X} := \mathbb{X} \cup \{v_i\}$;
11     $i$++;
12 return false;
```

Now we have

$$T_{\mathcal{C}_v} = \mathcal{L}(\mathcal{T}_{cub}) \cup \bigcup_{x \in \mathbb{X}} \mathcal{L}(\mathcal{T}^{sq}_{x,v}). \tag{23}$$

The lemma then follows by Proposition 1. □

To express solving a system of string constraints Φ in the terms of a (modified) RMC, we first convert Φ (using Algorithm 2) to an equisatisfiable cubic system Φ'. The satisfiability of a system of word equations Φ can be reduced to a modified RMC problem $(\mathcal{I}^{\wedge eq}_{\Phi}, \mathcal{T}^{v_i,\wedge eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{v_i,\wedge eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{\wedge eq})$ instantiating Algorithm 1 with the following components:

$$\mathcal{I}^{\wedge eq}_{\Phi} = \mathsf{encode}(\Phi') \qquad \mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \alpha x} = T_{\mathcal{C}_v} \circ \bigcup_{y \in \mathbb{X}, \alpha \in \Sigma_{\mathbb{X}}} T^{\wedge eq,3}_{y \rightarrowtail \alpha y} \qquad \mathcal{D}^{\wedge eq} = \left\{\left[\begin{smallmatrix}\square\\\square\end{smallmatrix}\right], \left[\begin{smallmatrix}\#\\\#\end{smallmatrix}\right]\right\}^*$$

$$\mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \epsilon} = T_{\mathcal{C}_v} \circ \bigcup_{y \in \mathbb{X}} T^{\wedge eq,3}_{y \rightarrowtail \epsilon}$$

For the modified RMC algorithm, we need to assume that $\mathbb{V} = \{v_1, v_2, \ldots\} \cap \Sigma_{\mathbb{X}} = \emptyset$, where $\mathbb{V}$ is a set of *fresh index variables*. We also need to update Line 3 of Algorithm 1 to $\mathcal{T}^v := \mathcal{T}^v_{x \rightarrowtail \alpha x} \cup \mathcal{T}^v_{x \rightarrowtail \epsilon}$ and Line 9 to $reach_{i+1} := \mathcal{T}^{v_i}(reach_i)$; $\mathbb{X} := \mathbb{X} \cup \{v_i\}$; to allow using a new variable v_i in every iteration (here, in every iteration of the algorithm, $\mathcal{T}^v$ will be instantiated with a new value of the v parameter). The entire algorithm is shown in Algorithm 3. Rationality of $\mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 1.

Lemma 10. *Algorithm 3 instantiated with $(\mathcal{I}^{\wedge eq}_{\Phi}, \mathbb{V}, \mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{\wedge eq})$ is sound if Φ is cubic.*

Proof. We again encode the nodes of a Nielsen proof graph as strings. The initial and final node correspond to strings from the encoded initial $\mathcal{I}^{\wedge eq}_{\Phi}$ and final language $\mathcal{D}^{\wedge eq}$, respectively. The relations $\mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{v,\wedge eq}_{x \rightarrowtail \epsilon}$ implement the Nielsen rules $x \rightarrowtail \alpha x$ and $x \rightarrowtail \epsilon$. For an arbitrary system of word equations the Nielsen proof graph may be infinite. However, since the transformation $\mathcal{C}_v$ preserves satisfiability, the procedure is sound. □

Completeness. Since the previous approach can in each step introduce a new equation (due to the transducer $T_{\mathcal{C}_v}$ transforming the system of equations to a cubic one), completeness is not guaranteed in general. In order to get a sound and complete procedure for cubic equations, it is necessary to use transducers with an increasing number of symbols being rewritten. In particular, we need to use transducers implementing the following relations:

$$\mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\alpha x} = \bigcup_{y\in\mathbb{X},\alpha\in\Sigma_{\mathbb{X}}} T^{\wedge eq,2^{i+1}}_{y\rightarrowtail\alpha y} \qquad \mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\epsilon} = \bigcup_{y\in\mathbb{X}} T^{\wedge eq,2^{i+1}}_{y\rightarrowtail\epsilon}$$

with $i \in \mathbb{N}_1$ being the number of iteration (moreover, it is also necessary to skip Line 10 in Algorithm 3). After each step, the maximum number of occurrences of a variable in the system is in the worst case multiplied by two. Therefore, in the ith step, the number of occurrence of a variable in the system can be up to 2^{i+1}, which can be handled by $\mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\epsilon}$.

Lemma 11. *Algorithm 3 instantiated with* $(\mathcal{I}^{\wedge eq}_{\Phi}, \mathbb{N}, \mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\alpha x} \cup \mathcal{T}^{i,\wedge eq\star}_{x\rightarrowtail\epsilon}, \mathcal{D}^{\wedge eq})$ *and modified as described above is sound and complete if* Φ *is cubic.*

Proof. As in previous cases, we encode the nodes of a Nielsen proof graph as strings. Since the transducer $T^{\wedge eq,\ell}_{y\rightarrowtail\alpha y}$ correctly implements the Nielsen rule $y \rightarrowtail \alpha y$ for systems where each variable occurs at most ℓ times, it suffices to show that in the ith iteration in Algorithm 3, the number of occurrences of each variable is bounded by 2^{i+1}. The soundness and completeness then follows from the BFS strategy of Algorithm 3 and Lemma 3.

Consider a system of equations Φ s.t. each variable occurs at most n times. Application of the rule of the form $y \rightarrowtail \epsilon$ does not increase the number of occurrences of y. The situation is different for a rule of the form $y \rightarrowtail \alpha y$ s.t. $\alpha \in \mathbb{X}$. Furthermore, observe that there are at most n occurrences of y and $(n-1)+(n-1) = 2n-2$ occurrences of α in the modified system (each y is replaced by αy and the first occurrence of α is removed from the modified system). The maximum number of occurrences of variables in the ith iteration of Algorithm 3 is hence bounded by 2^{i+1} (note that this is a loose upper bound, since the number of occurrences increases to $2n-2 \le 2n$ in the worst case), provided that Φ is cubic. □

Termination. Since the Nielsen transformation does not guarantee termination for the general case, neither does our algorithm. Investigation of possible symbolic encodings of complete algorithms, e.g. Makanin's algorithm (Makanin, 1977), is our future work.

5. Handling Boolean combination of string constraints

In this section, we will extend the procedure from handling a *conjunction* of word equations into a procedure that handles their arbitrary Boolean combination. An obvious approach is by combining the solutions we have given in Sections 3 and 4 with standard DPLL(T)-based solvers and use our procedure to handle the string theory. We can, however, solve the whole formula with our procedure by using the encoding proposed in this section. Although we do not have hope that the presented solution can compete with the highly-optimized DPLL(T)-based solvers, it (also taking into account the extensions from Section 6) makes our framework more robust, by having a homogeneous automata-based encoding for a quite general class of constraints. When one, e.g., tries to extend the approach by using abstraction (cf. Bouajjani et al. (2012)) to accelerate termination or by *learning* the invariant in the spirit of Neider and Jansen (2013), they can then still treat the encoding of the whole system of constraints uniformly within the framework of RMC.

The negation of word equations can be handled in the standard way. For instance, we can use the approach given by Abdulla et al. (2014) to convert a negated word equation $\mathsf{t}_\ell \neq \mathsf{t}_r$ to the following string constraint:

$$\bigvee_{c\in\Sigma}(\mathsf{t}_\ell = \mathsf{t}_r\cdot cx \vee \mathsf{t}_\ell\cdot cx = \mathsf{t}_r) \vee \bigvee_{c_1,c_2\in\Sigma,c_1\neq c_2}(\mathsf{t}_\ell = yc_1x_1 \wedge \mathsf{t}_r = yc_2x_2) \tag{24}$$

The first part of the constraint says that either t_ℓ is a strict prefix of t_r or the other way around. The second part says that t_ℓ and t_r have a common prefix y and start to differ in the next characters c_1 and c_2. For word equations connected using $\wedge$ and $\vee$, we apply distributive laws to obtain an equivalent formula in the conjunctive normal form (CNF) whose size is at worst exponential in the size of the original formula. Note that we cannot use the Tseitin transformation (Tseitin, 1983), since it may introduce fresh negated variables and their removal using Eq. (24) would destroy the CNF form.

Let us now focus on how to express solving a string constraint Φ composed of an arbitrary Boolean combination of word equations using a (modified) RMC. We start by removing inequalities in Φ using Eq. (24), then we convert the system without inequalities into CNF, and, finally, we apply the procedure in Lemma 8 to convert the CNF formula to an equisatisfiable and cubic CNF Φ'. For deciding satisfiability of Φ' in the terms of RMC, both the transition relations and the destination set remain the same as in the previous section (general case). The only difference is the initial configuration because the system is not a conjunction of terms any more but rather a general formula in CNF. For this, we extend the definition of encode to a clause $c : (\mathsf{t}^1_\ell = \mathsf{t}^1_r \vee \ldots \vee \mathsf{t}^n_\ell = \mathsf{t}^n_r)$ as $\mathsf{encode}(c) = \bigcup_{1\le j\le n} \mathsf{encode}(\mathsf{t}^j_\ell, \mathsf{t}^j_r)$. Then the initial configuration for Φ' is given as

$$\mathcal{I}^{\wedge\vee eq}_{\Phi'} = \mathsf{encode}(c_1).\left\{\begin{bmatrix}\#\\\#\end{bmatrix}\right\}.\ldots.\left\{\begin{bmatrix}\#\\\#\end{bmatrix}\right\}.\mathsf{encode}(c_m), \tag{25}$$

where Φ' is of the form $\Phi' : c_1 \wedge \ldots \wedge c_m$ and each clause $c_i : (\mathsf{t}^1_\ell = \mathsf{t}^1_r \vee \ldots \vee \mathsf{t}^{n_i}_\ell = \mathsf{t}^{n_i}_r)$. We obtain the following lemma directly from Proposition 1.

Lemma 12. *The initial set* $\mathcal{I}^{\wedge\vee eq}_{\Phi'}$ *is regular.*

The transition relation and the destination set are the same as the ones in the previous section, i.e., $\mathcal{T}^{v,\wedge\vee eq}_{x\rightarrowtail\alpha x} = \mathcal{T}^{v,\wedge eq}_{x\rightarrowtail\alpha x}$, $\mathcal{T}^{v,\wedge\vee eq}_{x\rightarrowtail\epsilon} = \mathcal{T}^{v,\wedge eq}_{x\rightarrowtail\epsilon}$, and $\mathcal{D}^{\wedge\vee eq} = \mathcal{D}^{\wedge eq}$. The soundness of our procedure for a Boolean combination of word equations is summarized by the following lemma. The completeness can be achieved, as in Section 4.2, by transducers with an increasing number of rewritten symbols.

Lemma 13. *Given a Boolean combination of word equations* Φ*, Algorithm 3 instantiated with* $(\mathcal{I}^{\wedge\vee eq}_{\Phi'}, \mathbb{V}, \mathcal{T}^{v,\wedge\vee eq}_{x\rightarrowtail\alpha x} \cup \mathcal{T}^{v,\wedge\vee eq}_{x\rightarrowtail\epsilon}, \mathcal{D}^{\wedge\vee eq})$ *is sound.*

Proof. A system of full word equations can be converted according to the steps described above to an equisatisfiable system in CNF $\Psi : \bigwedge^n_{i=1} c_i$ where every c_i is a disjunction of equalities. Then, Ψ is satisfiable if there is some $\phi : \bigwedge^n_{i=1} \mathsf{t}^i_\ell = \mathsf{t}^i_r$ where $(\mathsf{t}^i_\ell = \mathsf{t}^i_r) \in c_i$ for all i. Moreover, we have $\mathsf{encode}(\phi) \in \mathcal{I}^{\wedge\vee eq}_{\Phi}$. From Lemma 10 (and from the BFS strategy of RMC), we get that our algorithm is sound in proving Φ is satisfiable. □

Completeness is guaranteed if we consider the transducers $\mathcal{T}^{i,\wedge eq\star}$ for Φ in the same way as in Section 4.2. Regarding termination, it cannot be guaranteed, due to the corresponding result in Section 4.2 that holds for a special case of the Boolean combination of string equations we consider here.

6. Extensions

In this section, we discuss how to extend our RMC-based framework to support the following two types of *atomic constraints*:

(i) A *length constraint* φ_i is a formula of Presburger arithmetic over the values of $|x|$ for $x \in \mathbb{X}$, where $|\cdot| : \mathbb{X} \to \mathbb{N}$ is the word length function (to simplify the notation we use a formula of Presburger arithmetic with free variables $\mathbb{X}$ and we keep in mind that the value assigned to $x \in \mathbb{X}$ corresponds in fact to $|x|$).

(ii) A *regular constraint* φ_r is a conjunction of atoms of the form $x \in \mathcal{L}(\mathcal{A})$ (or their negation) where x is a word variable and $\mathcal{A}$ is an FA representing a regular language.

6.1. Length constraints

In order to extend our framework to solve word equations with length constraints, we encode them as regular languages, and we encode the effect of Nielsen transformations on the lengths of variables as regular relations. Let us start with defining *atomic length constraints*:

$$\varphi_{len} ::= a_1 x_1 + \cdots + a_n x_n \le c$$

for string variables $x_1, \ldots, x_n \in \mathbb{X}$ and integers $a_1, \ldots, a_n, c \in \mathbb{Z}$ (we will also use formulae in a less restricted form, which can always be translated to the defined one using standard arithmetic rules). Given a variable assignment $I : \mathbb{X} \to \Sigma^*$, it holds that I is a model of φ_{len}, written as $I \models \varphi_{len}$, iff $a_1 \cdot |I(x_1)| + \cdots + a_n \cdot |I(x_n)| \le c$. We note that the satisfiability of a string constraint with only atomic length constraints connected via Boolean connectives (i.e., no word equations) corresponds to the satisfiability of a Boolean combination of constraints in *integer linear programming*, which is an NP-complete problem (Karp, 1972).

We will show that length constraints can be encoded into our framework using standard automata-based techniques for dealing with constraints in Presburger arithmetic (Presburger, 1929; Wolper and Boigelot, 2000, 2002). First, let us define how a first-order variable ranging over $\mathbb{N}$ is represented in MSO(Str). Let $LSBF : \mathbb{N} \to 2^{\mathbb{N}}$ be a function representing the *least-significant bit first* binary encoding of a number such that for $n \in \mathbb{N}$, we define $LSBF(n)$ to be the finite set $S \subseteq \mathbb{N}$ for which $n = \sum_{i \in S} 2^i$. For instance, $LSBF(42) = \{1, 3, 5\}$ because $42 = 2^1 + 2^3 + 2^5$. Moreover, we define the *positional* least-significant bit first binary encoding of a number $n \in \mathbb{N}$ as $LSBF_p(n) = \{\ell + 1 \mid \ell \in LSBF(n)\}$.

Proposition 14. *Let $\varphi_{len}(x_1, \ldots, x_n)$ be an atomic length constraint. Then there exists an MSO(Str) formula $\psi_{len}(X_1, \ldots, X_n)$ with free position variables $X_1, \ldots, X_n$ such that an assignment $\sigma : \{x_1, \ldots, x_n\} \to \mathbb{N}$ is a model of φ_{len} iff the assignment $\sigma' = \{X_i \mapsto LSBF_p(v_i) \mid \sigma(x_i) = v_i\}$ is a model of ψ_{len}.*

Proof. A possible encoding of φ_{len} into ψ_{len} is given, e.g., in Glenn and Gasarch (1996). □

Recall that in automata-based approaches to Presburger arithmetic (such as Glenn and Gasarch (1996), Wolper and Boigelot (2000) and Wolper and Boigelot (2002)), a formula φ with k free variables is translated into an automaton $\mathcal{A}_\varphi$ over the alphabet $\mathbb{B}^k$ for $\mathbb{B} = \{0, 1\}$. A model of φ is represented as a word $w \in (\mathbb{B}^k)^*$ in the language of $\mathcal{A}_\varphi$ such that projecting the track for variable x_i from w gives us the $LSBF$ encoding of the value of x_i in the model. For instance, if $\begin{array}{c} x: \\ y: \end{array}\!\!\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \in \mathcal{L}(\mathcal{A}_\varphi)$, then the assignment $\{x \mapsto 9, y \mapsto 5\}$ is a model of φ because 1001 is a LSBF binary encoding of the number 9 and 1010 encodes the number 5.

In order to encode dealing with length constraints into our framework, Proposition 14 is not sufficient: we also need to be able to represent how the transformations modify those constraints and how the constraints restrict the space of possible solutions. In the following paragraphs, we provide the details about our approach.

Consider a word w_σ encoding an assignment $\sigma : \mathbb{X} \to \mathbb{N}$. The transformation $x \rightarrowtail yx$ for $x, y \in \mathbb{X}$ applied to w_σ produces a word $w_{\sigma'}$ encoding the assignment $\sigma' = \sigma \lhd \{x \mapsto \sigma(x) - \sigma(y)\}$ if $\sigma(x) \ge \sigma(y)$, where $\sigma' = \sigma \lhd \{x \mapsto n\}$ is defined as $\sigma'(x) = n$ and $\sigma'(y) = \sigma(y)$ for all $y \neq x$. The transformation $x \rightarrowtail ax$, for $a \in \Sigma$ produces a word $w_{\sigma'}$ encoding the assignment $\sigma' = \sigma \lhd \{x \mapsto \sigma(x) - 1\}$ if $\sigma(x) \ge 1$. Finally, the transformation $x \rightarrowtail \epsilon$ does not change the word w_σ, but imposes the restriction $\sigma(x) = 0$. Formally, the transformations are described using the following formulae:

$$\begin{aligned} \varphi^{len}_{x \rightarrowtail yx}(x, y, x') &\triangleq x \ge y \wedge x' = x - y, \\ \varphi^{len}_{x \rightarrowtail ax}(x, x') &\triangleq x \ge 1 \wedge x' = x - 1, \text{ and} \\ \varphi^{len}_{x \rightarrowtail \epsilon}(x) &\triangleq x = 0. \end{aligned} \tag{26}$$

From Propositions 14 and 2, it follows that the relations denoted by the formulae are regular. We will denote the transducers encoding those relations as $T^{len}_{x \rightarrowtail yx}$, $T^{len}_{x \rightarrowtail ax}$, and $T^{len}_{x \rightarrowtail \epsilon}$ respectively.

Let us now focus on how to adjust the initial and destination sets for an equation with a length constraint $\varphi_i(\mathbb{X})$ with free variables $\mathbb{X}$. The initial set is extended by all encoded models of φ_i. Formally, the part of the initial set related to the length constraint is given as $\mathcal{I}_{\varphi_i} = \mathcal{L}(\varphi_i)$ (which is a subset of $(\mathbb{B}^{|\mathbb{X}|})^*$) and the part of the destination set as $\mathcal{D}_{len} = (\mathbb{B}^{|\mathbb{X}|})^*$.

Satisfiability of a quadratic equation $eq : \mathrm{t}_\ell = \mathrm{t}_r$ with the length constraint φ_i can then be expressed as the RMC problem $(\mathcal{I}^{len}_{\varphi_i}, \mathcal{T}^{len}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{len}_{x \rightarrowtail \epsilon}, \mathcal{D}^{len}_{\varphi_i})$ instantiating Algorithm 1 with items given as follows (note the use of a fresh delimiter $\#_{len}$ for length constraints):

$$\mathcal{I}^{len}_{\varphi_i} = \mathcal{I}^{eq}.\{\#_{len}\}.\mathcal{I}_{\varphi_i} \qquad \mathcal{D}^{len}_{\varphi_i} = \mathcal{D}^{eq}.\{\#_{len}\}.\left(\mathbb{B}^{|\mathbb{X}|}\right)^*$$

$$\begin{aligned} \mathcal{T}^{len}_{x \rightarrowtail \alpha x} &= \bigcup_{y \in \mathbb{X}, z \in \mathbb{X}} T^{\le 2}_{y \rightarrowtail zy}.\{\#_{len} \mapsto \#_{len}\}.T^{len}_{y \rightarrowtail zy} \cup \\ &\quad \bigcup_{y \in \mathbb{X}, a \in \Sigma} T^{\le 2}_{y \rightarrowtail ay}.\{\#_{len} \mapsto \#_{len}\}.T^{len}_{y \rightarrowtail ay} \\ \mathcal{T}^{len}_{x \rightarrowtail \epsilon} &= \bigcup_{y \in \mathbb{X}} T^{\le 2}_{y \rightarrowtail \epsilon}.\{\#_{len} \mapsto \#_{len}\}.T^{len}_{y \rightarrowtail \epsilon} \end{aligned}$$

Extensions to a system of equations and (more generally) a Boolean combination of constraints can be done in the same manner as in Sections 4 and 5.

Rationality of $\mathcal{T}^{len}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{len}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 1. The soundness and completeness of our algorithm is summarized by Lemma 15. Termination is an open problem even for quadratic equations (cf. Büchi and Senger (1990)).

Lemma 15. *Given a quadratic word equation $eq : \mathrm{t}_\ell = \mathrm{t}_r$ with the length constraint φ_i, Algorithm 1 instantiated with $(\mathcal{I}^{len}_{\varphi_i}, \mathcal{T}^{len}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{len}_{x \rightarrowtail \epsilon}, \mathcal{D}^{len}_{\varphi_i})$ is sound and complete.*

Proof. We can generalize nodes of the Nielsen proof graph to pairs of the form $(\mathrm{t}'_\ell = \mathrm{t}'_r, f)$ where f is a mapping assigning lengths to variables from $\mathbb{X}$. The transformation rules can be straightforwardly generalized to take into account also the lengths. The initial nodes are pairs $(\mathrm{t}_\ell = \mathrm{t}_r, f)$ where f is a model of φ_i. The final nodes are nodes $(\epsilon = \epsilon, g)$ where g is arbitrary. Note that the generalized graph is not necessarily finite even for quadratic equations. Nevertheless, if the equation is satisfiable then there is a finite path from an initial node to a final node.

Directly from the definition of $\mathcal{I}^{len}_{\varphi_i}$ we have that the initial nodes of the generalized proof graph are encoded strings from $\mathcal{I}^{len}_{\varphi_i}$ and the final nodes correspond to $\mathcal{D}^{len}_{\varphi_i}$. We can also see that the transformation rules correspond to the encoded relations $\mathcal{T}^{len}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}^{len}_{x\rightarrowtail\epsilon}$. Since the search in Algorithm 1 implements a BFS strategy, we get that our (semi-)algorithm is sound and complete in proving satisfiability. $\square$

For the general (non-quadratic) case and the case of a Boolean combination of constraints, we can obtain a sound and complete (though non-terminating) procedure by using the transducers $\mathcal{T}^{i,\wedge eq\star}$ in the same way as in Section 4.2 and modifying the proof of Lemma 15 accordingly.

6.2. Regular constraints

Our second extension of the framework is the support of regular constraints as a conjunction of atoms of the form $x \in \mathcal{L}(\mathcal{A})$ for an FA $\mathcal{A}$ over Σ (note that the negation of an atom $x \notin \mathcal{L}(\mathcal{A})$ can be converted to the positive atom $x \in \mathcal{L}(\mathcal{A}^{\complement})$ where $\mathcal{A}^{\complement}$ is a complement of the FA $\mathcal{A}$). In particular, we assume that regular constraints are represented by a conjunction φ_r of ℓ atoms of the form

$$\varphi_r \triangleq \bigwedge_{i=1}^{\ell}(x_i \in \mathcal{L}(\mathcal{A}_i)), \tag{27}$$

where $\mathcal{A}_i$ is an FA for each $1 \le i \le \ell$. Without loss of generality, we assume that the automata occurring in φ_r have pairwise disjoint sets of states and, further, we use $\mathcal{A}_r = (Q_r, \Sigma, \delta_r, I_r, F_r)$ to denote the automaton constructed as the disjoint union of all automata occurring in formula φ_r. Note that the disjoint union of two FAs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, Q_i^1, Q_f^1)$ and $\mathcal{A}_1 = (Q_2, \Sigma, \Delta_2, Q_i^2, Q_f^2)$ is the FA $\mathcal{A}_1 \uplus \mathcal{A}_2 = (Q_1 \uplus Q_2, \Sigma, \Delta_1 \uplus \Delta_2, Q_i^1 \uplus Q_i^2, Q_f^1 \uplus Q_f^2)$.

The approach we developed here is inspired by the approach in Norn (cf. (Abdulla et al., 2015)), but the idea needed to be significantly modified to fit in our (more proof-based) framework. In particular, we encode regular constraints as words over symbols of the form $\langle x, p, q\rangle$ where $x \in \mathbb{X}$ and $p, q \in Q_r$. We denote the set of all such symbols as $\Gamma_{\mathbb{X},\mathcal{A}_r}$. Moreover, we treat the words as sets of symbols and hence we assume a fixed linear order $\preccurlyeq$ over symbols to allow a unique representation. In particular, for a word $w \in \Gamma^*_{\mathbb{X},\mathcal{A}_r}$ we use $w_{\preccurlyeq}$ to denote the string containing symbols sorted by $\preccurlyeq$ with no repetitions of symbols. A single atom $x \in \mathcal{L}(\mathcal{A}_i)$ for $\mathcal{A} = (Q_i, \Sigma, \delta_i, I_i, F_i)$ can be encoded as a set of words $\mathsf{encode}(x \in \mathcal{L}(\mathcal{A}_i)) = \{\langle x, p, q\rangle \mid p \in I_i, q \in F_i\}$. The set represents all possible combinations of initial and final states in $\mathcal{A}_i$. The initial set $\mathcal{I}_{\varphi_r}$ is then defined as

$$\mathcal{I}_{\varphi_r} = \{w_{\preccurlyeq} \in \Gamma^*_{\mathbb{X},\mathcal{A}_r} \mid w \in \mathsf{encode}(x_1 \in \mathcal{L}(\mathcal{A}_1))\dots \mathsf{encode}(x_\ell \in \mathcal{L}(\mathcal{A}_\ell))\}. \tag{28}$$

Note that $\mathcal{I}_{\varphi_r}$ is finite for a finite $\mathbb{X}$, therefore it is a regular language.

Let us now describe the effect of the Nielsen transformation on the regular constraint part. Consider a word w encoding a set of symbols from $\Gamma_{\mathbb{X},\mathcal{A}_r}$. Then, the transformation $x\rightarrowtail yx$ for $x, y \in \mathbb{X}$ applied to w produces words w' encoding sets where each occurrence of a symbol $\langle x, p, q\rangle$ is replaced with all possible pairs of symbols $\langle y, p, s\rangle$ and $\langle x, s, q\rangle$ where $p \rightsquigarrow s$ and $s \rightsquigarrow q$ in $\mathcal{A}_r$ (we use $q \rightsquigarrow q'$ to denote that there is a path from q to q' in the transition diagram of $\mathcal{A}_r$). Similarly, the transformation $x\rightarrowtail ax$ for $x \in \mathbb{X}, a \in \Sigma$ applied to w produces words w' encoding sets where each occurrence of a symbol $\langle x, p, q\rangle$ is replaced with all possible symbols $\langle x, r, q\rangle$ where $p\xrightarrow{a}r$ in $\mathcal{A}_r$. Finally, by the transformation $x\rightarrowtail\epsilon$ we obtain a string $w' = w$ only if all symbols of w related to the variable x are of the form $\langle x, q, q\rangle$ for $q \in Q$. Formally, we first define the function expanding a single symbol for variables x and y as

$$exp_{x,y}(\sigma) = \begin{cases}\{\,\langle y, p, r\rangle.\langle x, r, q\rangle \mid p \rightsquigarrow r \rightsquigarrow q \text{ in } \mathcal{A}_r\,\} & \text{if } \sigma = \langle x, p, q\rangle,\\ \{\sigma\} & \text{otherwise.}\end{cases} \tag{29}$$

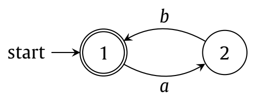
Similarly, we define the expansion

$$exp_{x,a}(\sigma) = \begin{cases}\{\,\langle x, r, q\rangle \mid p\xrightarrow{a}r \rightsquigarrow q \text{ in } \mathcal{A}_r\,\} & \text{if } \sigma = \langle x, p, q\rangle,\\ \{\sigma\} & \text{otherwise.}\end{cases} \tag{30}$$

Then, the transformations $x\rightarrowtail yx$, $x\rightarrowtail ax$, and $x\rightarrowtail\epsilon$ can be described by the following relations:

$$\begin{aligned} T^{reg}_{x\rightarrowtail yx} &= \{\,(w, u_{\preccurlyeq}) \mid u \in exp_{x,y}(w[1])\dots exp_{x,y}(w[|w|])\,\},\\ T^{reg}_{x\rightarrowtail ax} &= \{\,(w, u_{\preccurlyeq}) \mid u \in exp_{x,a}(w[1])\dots exp_{x,a}(w[|w|])\,\}, \text{ and}\\ T^{reg}_{x\rightarrowtail \epsilon} &= \Big\{\ (w, w) \mid \forall 1 \le i \le |w| :\\ &\qquad \forall p, q \in Q : w[i] = \langle x, p, q\rangle \Rightarrow p = q\ \Big\}. \end{aligned} \tag{31}$$

Example 7. Consider the regular constraint $x \in \mathcal{L}(\mathcal{A}_x)$ with $\mathcal{A}_x$ given below.



Then, the corresponding values will be as follows:

$\mathsf{encode}(x \in \mathcal{L}(\mathcal{A}_x)) = \{\langle x, 1, 1\rangle\}$

$exp_{x,y}(\langle x, 1, 1\rangle) = \{\langle y, 1, 1\rangle.\langle x, 1, 1\rangle,\ \langle y, 1, 2\rangle.\langle x, 2, 1\rangle\}$

$exp_{x,a}(\langle x, 1, 1\rangle) = \{\langle x, 2, 1\rangle\}$

Moreover, $T^{reg}_{x\rightarrowtail yx}$ will, e.g., contain pairs $(\langle x, 1, 1\rangle.\langle y, 1, 2\rangle, u_{\preccurlyeq})$ with

$u_{\preccurlyeq} \in \{\langle x, 1, 1\rangle.\langle y, 1, 1\rangle.\langle y, 1, 2\rangle,\ \langle x, 2, 1\rangle.\langle y, 1, 2\rangle\}$

(we assume that $\preccurlyeq$ is a lexicographic ordering on the components). Note that symbols in the words are sorted by $\preccurlyeq$ with duplicates removed. Similarly, $T^{reg}_{x\rightarrowtail ax}$ will, e.g., contain the pair $(\langle x, 1, 1\rangle.\langle y, 1, 2\rangle,\ \ \langle x, 2, 1\rangle.\langle y, 1, 2\rangle)$, and $T^{reg}_{x\rightarrowtail\epsilon}$ will contain $(\langle x, 1, 1\rangle.\langle y, 1, 2\rangle,\ \langle x, 1, 1\rangle.\langle y, 1, 2\rangle)$. $\square$

The following lemma shows that the transformations are rational. In the proof, we first construct MSO(Str) formulae realizing necessary set operations on strings and the effect of the expanding function. Based on them, we construct formulae realizing the transformations, by means of the relation $\mathsf{pad}_{\square}(T)$ appending to w and w' with $(w, w') \in T$ an arbitrary number of symbols $\square$ (cf. Section 3.4).

Lemma 16. *The relations* $\mathsf{pad}_{\square}(T^{reg}_{x\rightarrowtail yx})$, $\mathsf{pad}_{\square}(T^{reg}_{x\rightarrowtail ax})$, *and* $\mathsf{pad}_{\square}(T^{reg}_{x\rightarrowtail\epsilon})$ *are rational.*

Proof. In this proof, we extend the total order $\preccurlyeq$ on $\Gamma_{\mathbb{X},\mathcal{A}_r}$ to a total order on $\Gamma_{\mathbb{X},\mathcal{A}_r} \cup \{\square\}$ such that $\forall\sigma \in \Gamma_{\mathbb{X},\mathcal{A}_r} : \sigma \preccurlyeq \square$. (We note that encoding $\preccurlyeq$ in MSO(Str) is simple.) We define the relations $T^{reg}_{x\rightarrowtail yx}$ and $T^{reg}_{x\rightarrowtail\epsilon}$ using MSO(Str). The relation $T^{reg}_{x\rightarrowtail ax}$ can be defined analogously to $T^{reg}_{x\rightarrowtail yx}$.

$$\begin{aligned}\psi^{reg}_{x\rightarrowtail yx}(w, w') \triangleq\quad \exists^{\mathbb{W}} u_1, u_2, u_3 \Big(& filter_x(u_1, u_2, w) \wedge\\ & expand^y_x(u_1, u_3) \wedge\\ & union(u_2, u_3, w') \wedge ordSet(w')\Big)\end{aligned} \tag{32}$$

$$\psi^{reg}_{x\rightarrowtail\epsilon}(w,w') \triangleq \forall^{\mathbb{P}} i\Big(w[i]=w'[i] \land \bigvee_{\substack{\xi\in\Gamma_{\mathbb{X}\setminus\{x\},\mathcal{A}_r}\cup\\ \{(x,q,q)\mid q\in Q_r\}}} w'[i]=\xi\Big) \tag{33}$$

where $\mathit{filter}_x(u,v,w)$ partitions the symbols of w to u and v such that u contains symbols that are of the form $\langle x,-,-\rangle$, i.e., of the form $\langle x,q,s\rangle$ for arbitrary q and s, and v contains the remaining ones; $\mathit{expand}^y_x(u,v)$ replaces each symbol $\langle x,p,q\rangle$ in u with $\langle y,p,s\rangle$ and $\langle x,s,q\rangle$ in v; and *union* is a set-like union. These predicates (including auxiliary predicates) are defined as follows:

$$\sigma\in w \triangleq \exists^{\mathbb{P}} i(w[i]=\sigma) \tag{34}$$

$$\mathit{set}(u) \triangleq \neg\exists^{\mathbb{P}} i,j(i\neq j\land u[i]=u[j]) \tag{35}$$

$$\mathit{ordSet}(u) \triangleq \mathit{set}(u)\land\forall^{\mathbb{P}} i,j(i<j\to u[i]\preccurlyeq u[j]) \tag{36}$$

$$\begin{aligned}&\mathit{filter}_x(u,v,w)\triangleq\\ &\forall^{\mathbb{P}} i\Bigg(\bigwedge_{q,s\in Q_r}(w[i]=\langle x,q,s\rangle\to(u[i]=\langle x,q,s\rangle\land v[i]=\square))\\ &\land\bigwedge_{\substack{z\in\mathbb{X}\setminus\{x\}\\ q,s\in Q_r}}(w[i]=\langle z,q,s\rangle\to(u[i]=\square\land v[i]=\langle z,q,s\rangle))\Bigg)\end{aligned} \tag{37}$$

$$\mathit{expand}^y_x(u,v)\triangleq \tag{38}$$

$$\bigwedge_{\substack{s,q\in Q_r,s\rightsquigarrow q\\ \xi'=\langle x,s,q\rangle}}\Big(\xi'\in v\to\bigvee_{\substack{p\in Q_r,p\rightsquigarrow s\\ \xi=\langle x,p,q\rangle\\ \xi''=\langle y,p,s\rangle}}\xi\in u\land\xi''\in v\Big)\land \tag{39}$$

$$\bigwedge_{\substack{p,s\in Q_r,p\rightsquigarrow s\\ \xi''=\langle y,p,s\rangle}}\Big(\xi''\in v\to\bigvee_{\substack{q\in Q_r,s\rightsquigarrow q\\ \xi=\langle x,p,q\rangle\\ \xi'=\langle x,s,q\rangle}}\xi\in u\land\xi'\in v\Big)\land \tag{40}$$

$$\bigwedge_{\substack{p,q\in Q_r\\ \xi=\langle x,p,q\rangle}}\Big(\xi\in u\to\bigvee_{\substack{s\in Q_r,p\rightsquigarrow s\rightsquigarrow q\\ \xi''=\langle y,p,s\rangle\\ \xi'=\langle x,s,q\rangle}}\xi''\in u\land\xi'\in v\Big) \tag{41}$$

$$\mathit{union}(u,v,w)\triangleq\bigwedge_{\xi\in\Gamma_{\mathbb{X},\mathcal{A}_r}}\xi\in w\leftrightarrow(\xi\in u\lor\xi\in v) \tag{42}$$

Intuitively, in the definition of $\mathit{filter}_x(u,v,w)$, the first part picks from w symbols containing x and adds them into u and the second part picks from w the other symbols and adds them into v. On the other hand, in the definition of $\mathit{expand}^y_x(u,v)$, we use s to denote the *splitting* state on the path from state p to state q. Then, the first and the second parts of the formula denote that $\langle y,p,s\rangle$ and $\langle x,s,q\rangle$ are in v while the last part denotes that $\langle x,p,q\rangle$ is in u.

We further consider the relations $\tau_{+\mathsf{pad}}=\{(w,w')\mid w\in(\Gamma_{\mathbb{X},\mathcal{A}_r}\cup\{\square\})^*, w'\in w.\{\square\}^*\}$ and $\tau_{-\mathsf{pad}}=\{(w,w')\mid w'\in(\Gamma_{\mathbb{X},\mathcal{A}_r}\cup\{\square\})^*, w\in w'.\{\square\}^*\}$ appending and removing padding, respectively. These relations are rational. Then, observe that $\mathsf{pad}_\square(T^{reg}_{x\rightarrowtail yx})=\tau_{+\mathsf{pad}}\circ\tau_{-\mathsf{pad}}\circ\mathcal{L}(\psi^{reg}_{x\rightarrowtail yx})\circ\tau_{+\mathsf{pad}}$. Recall the relation $\mathsf{pad}_\square(T)$ appends to w' in $(w,w')\in T$ an arbitrary number of symbols $\square$ (cf. Section 3.4). From Propositions 1 and 2, we have that $\mathsf{pad}_\square(T^{reg}_{x\rightarrowtail yx})$ is rational (the same for $\mathsf{pad}_\square(T^{reg}_{x\rightarrowtail\epsilon})$). □

The last missing piece is a definition of the destination set containing all satisfiable regular constraints. For a variable $x\in\mathbb{X}$, we define the set of satisfiable x-constraints as $L^x=\{w_\preccurlyeq\mid w=\langle x,q_1,r_1\rangle\cdots\langle x,q_n,r_n\rangle\in\Gamma^*_{\mathbb{X},\mathcal{A}_r},\bigcap_{i=1}^n\mathcal{L}_{\mathcal{A}_r}(q_i,r_i)\neq\emptyset\}$. Then, the destination set for a set of variables $\mathbb{X}=\{x_1,\dots,x_k\}$ is given as $\mathcal{D}_{reg}=\{w_\preccurlyeq\mid w\in L^{x_1}\cdots L^{x_k}\}$. As in the case of $\mathcal{I}_{\varphi_r}$, the set $\mathcal{D}_{reg}$ is finite and hence regular as well.

Satisfiability of a quadratic word equation $eq\colon \mathsf{t}_\ell=\mathsf{t}_r$ with a regular constraint φ_r can be expressed in the RMC framework

Table 1
Summary of the proposed approach on various types of string constraints with extensions. S stands for *sound*, C stands for *complete*, and T stands for *terminating*.

	Quadratic system	*General system*	*Boolean combination*
Pure	SCT (Section 4.1)	SC (Section 4.2)	SC (Section 5)
Length	SC (Section 6.1)	SC (Section 6.1)	SC (Section 6.1)
Regular	SCT (Section 6.2)	SC (Section 6.2)	SC (Section 6.2)

as $(\mathcal{I}^{reg}_{\varphi_r},\mathcal{T}^{reg}_{x\rightarrowtail\alpha x}\cup\mathcal{T}^{reg}_{x\rightarrowtail\epsilon},\mathcal{D}^{reg}_{\varphi_r})$ instantiating Algorithm 1 with items given in as follows (note that we use a fresh delimiter $\#_{reg}$):

$$\mathcal{I}^{reg}_{\varphi_r}=\mathcal{I}^{eq}.\{\#_{reg}\}.\mathsf{pad}_\square(\mathcal{I}_{\varphi_r})\qquad\qquad \mathcal{D}^{reg}_{\varphi_r}=\mathcal{D}^{eq}.\{\#_{reg}\}.\mathsf{pad}_\square(\mathcal{D}_{reg})$$

$$\begin{aligned}\mathcal{T}^{reg}_{x\rightarrowtail\alpha x}&=\bigcup_{y\in\mathbb{X},z\in\mathbb{X}}T^{\le 2}_{y\rightarrowtail zy}.\{\#_{reg}\mapsto\#_{reg}\}.\mathsf{pad}_\square(T^{reg}_{y\rightarrowtail zy})\cup\\ &\quad\bigcup_{y\in\mathbb{X},a\in\Sigma}T^{\le 2}_{y\rightarrowtail ay}.\{\#_{reg}\mapsto\#_{reg}\}.\mathsf{pad}_\square(T^{reg}_{y\rightarrowtail ay})\\ \mathcal{T}^{reg}_{x\rightarrowtail\epsilon}&=\bigcup_{y\in\mathbb{X}}T^{\le 2}_{y\rightarrowtail\epsilon}.\{\#_{reg}\mapsto\#_{reg}\}.\mathsf{pad}_\square(T^{reg}_{y\rightarrowtail\epsilon})\end{aligned}$$

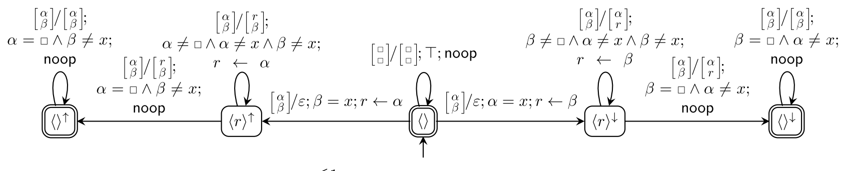
The rationality of $\mathcal{T}^{reg}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}^{reg}_{x\rightarrowtail\epsilon}$ follows directly from Proposition 1. The soundness and completeness of our procedure is summarized by the following lemma.

Lemma 17. *Given a quadratic word equation $eq\colon \mathsf{t}_\ell=\mathsf{t}_r$ with a regular constraint φ_r, Algorithm 1 instantiated with $(\mathcal{I}^{reg}_{\varphi_r},\mathcal{T}^{reg}_{x\rightarrowtail\alpha x}\cup\mathcal{T}^{reg}_{x\rightarrowtail\epsilon},\mathcal{D}^{reg}_{\varphi_r})$ is sound, complete, and terminating.*

Proof. Similarly to proof of Lemma 15, we can generalize nodes of the Nielsen proof graph to pairs of the form $(\mathsf{t}'_\ell=\mathsf{t}'_r,S)$ where $S\subseteq\Gamma_{\mathbb{X},\mathcal{A}_r}$. The transformation rules can be straightforwardly generalized to take into account also the regular constraints represented by a subset of $\Gamma_{\mathbb{X},\mathcal{A}_r}$. Since $\Gamma_{\mathbb{X},\mathcal{A}_r}$ is finite and *eq* is quadratic, the generalized proof graph is finite. The initial nodes of the generalized proof graph are exactly encoded strings from $\mathcal{I}^{reg}_{\varphi_r}$, the final nodes correspond to $\mathcal{D}^{reg}_{\varphi_r}$, and the transformation rules correspond to the encoded relations $\mathcal{T}^{reg}_{x\rightarrowtail\alpha x}$ and $\mathcal{T}^{reg}_{x\rightarrowtail\epsilon}$. Since our RMC framework implements the BFS strategy, from the previous we get that our procedure is sound, complete, and terminating in proving satisfiability. □

As in the case of length constraints, the satisfiability of a word equation *eq* with regular constraints can be generalized to a system of equations Φ with regular constraints. The languages/relations corresponding to *eq* are replaced by languages/relations corresponding to Φ. For a system of word equations with regular constraints our algorithm is still sound (and complete if we consider the transducers $\mathcal{T}^{\mathsf{i},\wedge eq\star}$ for Φ).

Discussion. In the previous sections, we elaborated the proposed RMC framework for various kinds of string constraints including length and regular extensions. In Table 1, we give a summary of the achieved results. If the system of equations is quadratic (and even enriched with regular extensions), then our RMC approach is sound, complete, and terminating. It is basically due to the fact that the language *processed* in Algorithm 1 for the transformations tailored for quadratic equations becomes saturated after a finite number of steps. In all other cases, our RMC approach is sound and complete (but not generally terminating). For suitable encoded transformations, we are able to reach a solution after a finite number of steps (if the system is satisfiable). But in general, the language *processed* in Algorithm 3 for these transformations is not guaranteed to eventually become saturated (see Fig. 11).


(a) An FRT for the relation $T^{\leq 1}_{x \mapsto \epsilon}$ for an arbitrary set of variables $\mathbb{X}$ and alphabet Σ

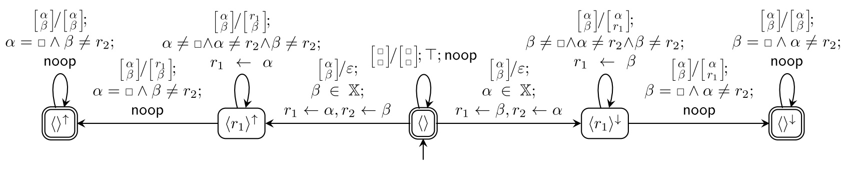

(b) An FRT for the relation $\bigcup_{x \in \mathbb{X}} T^{\leq 1}_{x \mapsto \epsilon}$ for an arbitrary set of variables $\mathbb{X}$ and alphabet Σ

Fig. 11. FRT (a) implementing the relation $T^{\leq 1}_{x \mapsto \epsilon}$ for an arbitrary set of variables $\mathbb{X}$ and alphabet Σ. The register r is used to store a shifted symbol. FRT (b) implementing the relation $\bigcup_{x \in \mathbb{X}} T^{\leq 1}_{x \mapsto \epsilon}$ for an arbitrary set of variables $\mathbb{X}$ and alphabet Σ. The register r_1 is used to store a shifted symbol and the register r_2 is used to store a variable that should be removed. In the figures, α and β are variables representing the input symbol, r, r_1, and r_2 are registers, and the transitions are of the form *action;condition;register update*.

7. Implementation

We created a prototype Python tool called Retro,[1] where we implemented the symbolic procedure for solving systems of word equations. Retro implements a modification of the RMC loop from Algorithm 1. In particular, instead of standard transducers defined in Section 2, it uses the so-called *finite-alphabet register transducers* (FRTs), which allow a more concise representation of a rational relation.

Informally, an FRT is a register automaton (in the sense of Kaminski and Francez (1994) and Demri and Lazic (2009)) where the alphabet is finite. The finiteness of the alphabet implies that the expressive power of FRTs coincides with the class of rational languages, but the advantage of using FRTs is that they allow a more concise representation than ordinary transducers. One can consider FRTs to be the restriction of symbolic transducers with registers of Veanes et al. (2012) to finite alphabets. Operations for dealing with these transducers are then straightforward restrictions of the operations considered by Veanes et al. (2012) and therefore do not elaborate on it here.

In particular, transducers (without registers) corresponding to the transformers $\mathcal{T}_{x \mapsto \alpha x}$ and $\mathcal{T}_{x \mapsto \epsilon}$ contain branching at the beginning for each choice of x and α. Especially in the case of huge alphabets, this yields huge transducers (consider for instance the Unicode alphabet with over 1 million symbols). The use of FRTs yields much smaller automata because the choice of x and α is stored into registers and then processed symbolically. To illustrate the effect of using registers, consider the transducer shown in Fig. 8 implementing the encoded relation $T^{\leq 1}_{x \mapsto \epsilon}$ for $\mathbb{X} = \{x\}$ and $\Sigma = \{a\}$. The full transducer for large alphabets would require a branching for each $\left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right]$, with $u, v \in \Sigma_{\mathbb{X}}$, and a lot of states to store the concrete shifted symbols. In particular, it requires at least one pair of states $\langle v \rangle^{\uparrow}$ and $\langle v \rangle^{\downarrow}$ for each $v \in \Sigma_{\mathbb{X}}$, which is unfeasible for very large $\Sigma_{\mathbb{X}}$. On the other hand, the FRT in Fig. 11(a) stores the shifted symbols in the register r, the branching is replaced by a symbolic transition, and hence it requires just a couple of states and transitions. Moreover, using an additional register r_2 to store the variable to replace, we are able to efficiently represent the relation $\bigcup_{x \in \mathbb{X}} T^{\leq 1}_{x \mapsto \epsilon}$, as shown in Fig. 11(b).

Concretely, Retro implements the decision procedure for a system of general word equations and length constraints (i.e., the procedures covered in Sections 3, 4 and 6.1). It does not implement (i) Boolean combinations of constraints (Section 5) and (ii) regular constraints (Section 6.2), which are quite inefficient and are provided in order for our approach to have a more robust theoretical formal basis for a large fragment of input constraints (cf. the discussion at the beginning of Section 5).

[1] available at https://github.com/VeriFIT/retro.

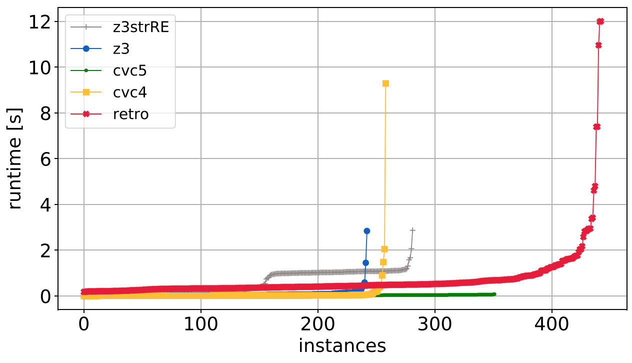


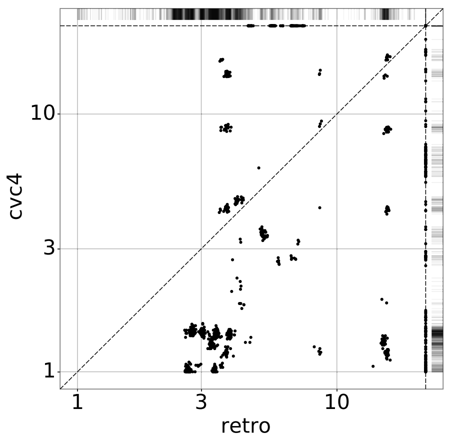
Fig. 12. A cactus plot comparing Retro, CVC4, CVC5, Z3, and Z3Str3RE on $\texttt{Kepler}_{22}$.

As another feature, Retro uses deterministic finite automata to represent configurations in Algorithm 1. It also uses eager automata minimization, since it has a big impact on the performance, especially on checking the termination condition of the RMC algorithm, which is done by testing language inclusion between the current configuration and all so-far processed configurations.

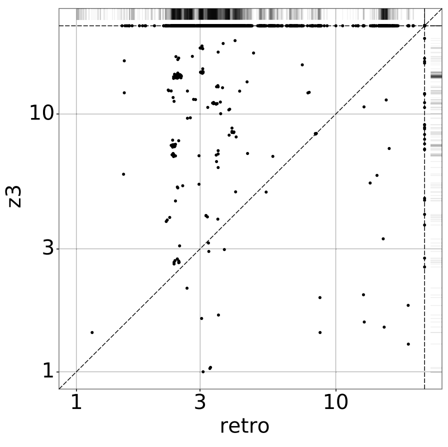
8. Experimental evaluation

We compared the performance of our approach (implemented in Retro) with four current state-of-the-art SMT solvers that support the string theory: Z3 4.8.14 (de Moura and Bjørner, 2008), Z3Str3RE (Berzish et al., 2021), CVC4 1.8 (Barrett et al., 2011), and CVC5 1.0.1 (Barbosa et al., 2022). Regarding other solvers that we are aware of, the performance of Norn from Abdulla et al. (2015) and Ostrich from Chen et al. (2019b) was much worse than the considered tools, the performance of Z3Str4 (Mora et al., 2021) was similar to that of Z3Str3RE, and Sloth of Holík et al. (2018) was unsound on the considered fragment (it supports only the so-called *straight-line fragment*) (see Fig. 13)

The first set of benchmarks is $\texttt{Kepler}_{22}$, obtained from Le and He (2018). $\texttt{Kepler}_{22}$ contains 600 hand-crafted string constraints composed of quadratic word equations with length constraints. In Fig. 12, we give a cactus plot of the results of the solvers



(a) Retro vs CVC4



(b) Retro vs Z3

Fig. 13. Comparison of Retro with CVC4 and Z3 on PyEx-Hard. We show difficult instances that took more than 1 s to finish. Times are given in seconds, axes are logarithmic.

on the $\texttt{Kepler}_{22}$ benchmark set with the timeout of 20 s. In cactus plots, the closer a solver's plot is to the right and bottom borders, the better is the corresponding solver. The total numbers of solved benchmarks within the timeout were: 243 for Z3, 282 for Z3Str3RE, 259 for CVC4, 352 for CVC5, and 443 for Retro. We can refine these numbers by comparing the cases solved by each pair of tools, as shown in Table 2. From the table we can see that Retro solves significantly more benchmarks than all other state-of-the-art tools, from 92 when compared with CVC5 to 201 with Z3. Only in one case Retro failed (as did Z3Str3RE) while CVC4, CVC5, and Z3 succeeded. Except for CVC4 vs. CVC5, the comparison between the other tools is inconclusive, as the entries (X, Y) and (Y, X) of the table both contain large values.

The other set of benchmarks that we tried is PyEx-Hard. Here we wanted to see the potential of integrating Retro with DPLL(T)-based string solvers, like Z3 or CVC4, as a specific string theory solver. The input of this component is a conjunction of atomic string formulae (e.g., $xy = zb \land z = ax$) that is a model of the Boolean structure of the top-level formula. The conjunction of atomic string formulae is then, in several layers, processed by various string theory solvers, which either add more conflict clauses or return a model. To evaluate whether Retro is suitable to be used as "one of the layers" of Z3 or CVC4's string solver, we analyzed the PyEx benchmarks from Reynolds et al. (2017) and extracted from it 967 difficult instances that neither CVC4 nor Z3 could solve in 10 s. From those instances, we obtained 20,020 conjunctions of word equations that Z3's DPLL(T) algorithm sent to its string theory solver when trying to solve them. We call those 20,020 conjunctions of word equations PyEx-Hard. We then evaluated the other solvers on PyEx-Hard with the timeout of 20 s. Out of these, Z3 could not solve 3001, Z3Str3RE 814, CVC4 152, CVC5 171, and Retro could not solve 3079 instances.

Let us now look closely at the hard instances in the PyEx-Hard benchmark set, in particular, on the instances that the other tools could not solve. These benchmarks cannot be handled by the (several layers of) fast heuristics implemented in these tools, which are sufficient to solve many benchmarks without the need to start applying the case-split rule.[2] In Fig. 13 we give a comparison of

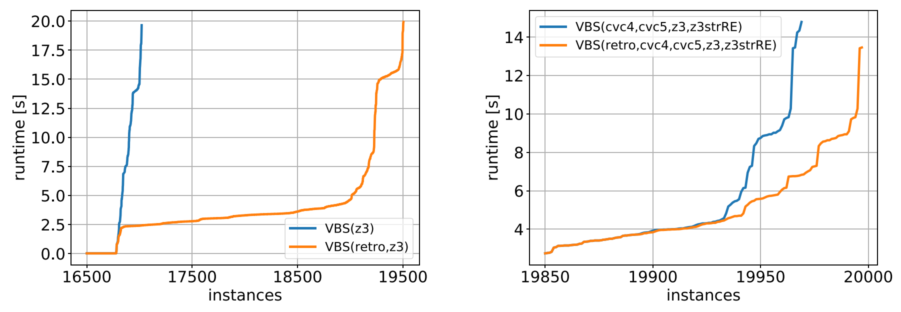
Table 2
A break-down of solved cases on the $\texttt{Kepler}_{22}$ benchmark. A number on row X in column Y denotes the number of cases that solver X could solve and solver Y could not solve.

	Retro	CVC4	CVC5	Z3	Z3Str3RE
Retro	–	185	92	201	161
CVC4	1	–	2	104	63
CVC5	1	95	–	197	145
Z3	1	88	88	–	23
Z3Str3RE	0	86	75	62	–

the running times of Retro with CVC4 and Z3 with a particular focus on the difficult instances (running time above 1 s). The plots about CVC5 and Z3Str3RE show similar trends. From the figure we can see that on many hard instances Retro can provide the answer much faster than the other tool, in particular for Z3. When we compare the solvers on the examples that the other tools failed to solve, we have that Retro could solve 86 examples (56.2%) out of those where CVC4 failed, 130 examples (76.02%) where CVC5 failed, 2484 examples (82.7%) where Z3 failed, and 519 examples (63.75%) where Z3Str3RE failed. Moreover, Retro solved 28 instances as the only tool. Lastly, we consider how Retro affects the *Virtual Best Solver*: given a set of solvers S, we use $VBS(S)$ to denote the solver that would be obtained by taking, for each benchmark, the fastest solver on the given benchmark. In Fig. 14, we provide cactus plots showing the impact of Retro on two instances of Virtual Best Solvers; the plots for Z3Str3RE, CVC4, and CVC5 are similar, with the one for Z3Str3RE being closer to Fig. 14(a) while the ones for CVC4/5 being closer to Fig. 14(b). As we can see, the plots show that our approach can significantly help solvers deal with hard equations.

Discussion. From the obtained results, we see that our approach works well in *difficult cases*, where the fast heuristics implemented in state-of-the-art solvers are not sufficient to quickly discharge a formula, which happens in particular when the (un)satisfiability proof is complex. Our approach can exploit the symbolic representation of the proof tree and use it to reduce the redundancy of performing transformations. Note that we can still beat the heavily optimized Z3, Z3Str3RE, CVC4, and CVC5 written in C++ by a Python prototype in those cases. We believe that

[2] For instance, when Z3 receives the word equation $xy = yax$, it infers the length constraint $|x| + |y| = |y| + 1 + |x|$, which implies unsatisfiability of the word equation without the need to start applying the case-split rule at all.



(a) VBS(Retro, Z3) (b) VBS(Retro, CVC4, CVC5, Z3, Z3Str3RE)

Fig. 14. A cactus plot comparing the Virtual Best Solver of (a) Z3 with and without Retro and (b) all tools with and without Retro on the PyEx-Hard benchmark. We show only the most difficult benchmarks (out of 20,020).

implementing our symbolic algorithm as a part of a state-of-the-art SMT solver would push the applicability of string solving even further, especially for cases of string constraints with a complex structure, which need to solve multiple DPLL(T) queries in order to establish the (un)satisfiability of a string formula.

9. Related work

The study of solving string constraint traces back to 1946, when (Quine, 1946) showed that the first-order theory of word equations is undecidable. Makanin (1977) achieved a milestone result by showing that the class of quantifier-free word equation is decidable. Since then, several works, e.g., Plandowski (1999, 2006), Matiyasevich (2008), Robson and Diekert (1999), Schulz (1990), Ganesh et al. (2012), Ganesh and Berzish (2016), Abdulla et al. (2014), Barceló et al. (2013), Lin and Barceló (2016), Chen et al. (2018, 2019b) and Abdulla et al. (2019), consider the decidability and complexity of different classes of string constraints. Efficient solving of satisfiability of string constraints is a challenging problem. Moreover, decidability of the problem of satisfiability of word equations combined with length constraints of the form $|x| = |y|$ has already been open for over 20 years (Büchi and Senger, 1990).

The strong practical motivation led to the rise of several string constraint solvers that concentrate on solving practical problem instances. The typical procedure implemented within *DPLL(T)-based* string solvers is to split the constraints into simpler subcases based on how the solutions are aligned, combining with powerful techniques for Boolean reasoning to efficiently explore the resulting exponentially-sized search space. The case-split rule is usually performed explicitly. Examples of solvers implementing this approach are Norn (Abdulla et al., 2014, 2015), Trau (Abdulla et al., 2018), Ostrich (Chen et al., 2019b), Sloth (Holík et al., 2018), CVC4 (Barrett et al., 2011), CVC5 (Barbosa et al., 2022), Z3str2 (Zheng et al., 2017), Z3str3 (Berzish et al., 2017), Z3str4 (Mora et al., 2021), Z3Str3RE (Berzish et al., 2021), S3 (Trinh et al., 2014), S3P (Trinh et al., 2016). In contrast, our approach performs case-splits symbolically.

Automata and transducers have been used in many approaches and tools for string solving, such as in Norn (Abdulla et al., 2014, 2015), Trau (Abdulla et al., 2018), Ostrich (Chen et al., 2019b), Sloth (Holík et al., 2018), Slog (Wang et al., 2016), Slent (Wang et al., 2018), or Z3Str3RE (Berzish et al., 2021), and also in string solvers for analyzing string-manipulating programs, such as ABC (Aydin et al., 2018) and Stranger (Yu et al., 2010), which soundly over-approximate string constraints using transducers (Yu et al., 2016). The main difference of these approaches to ours is that they use transducers to encode possible models (solutions) to the string constraints, while we use automata and transducers to encode the string constraint transformations. Other approaches for solving string constraints include reducing the constraints to the SMT theory of bit vectors (e.g., Zheng et al., 2017; Berzish et al., 2017; Mora et al., 2021; Kiezun et al., 2012), the theory of arrays (e.g., Li and Ghosh (2013)), or SAT-solving (e.g., Day et al., 2019; Amadini et al., 2017; Scott et al., 2017). Not so many approaches are based on algebraic approaches, such as the Nielsen transformation. In addition to our approach, it is also used as the basis of the work of Le and He (2018). On the other hand, the Nielsen transformation (Nielsen, 1917) is used by some tools that implement different approaches to discharge quadratic equations (e.g., Ostrich of Chen et al. (2019b)). Complex rewriting rules are used, e.g., when dealing with regular constraints in CVC5 (Nötzli et al., 2022).

CRediT authorship contribution statement

Yu-Fang Chen: Conceptualization, Methodology, Writing. **Vojtěch Havlena:** Conceptualization, Methodology, Writing, Software. **Ondřej Lengál:** Conceptualization, Methodology, Writing. **Andrea Turrini:** Conceptualization, Methodology, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank Mohamed Faouzi Atig for discussing the topic. This work has been partially supported by the National Natural Science Foundation of China (grant no. 61836005), the Chinese Academy of Sciences Project for Young Scientists in Basic Research (grant no. YSBR-040), the Czech Science Foundation project 19-24397S, the FIT BUT internal project FIT-S-20-6427, the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, and the project of Ministry of Science and Technology, Taiwan (grant no. MOST-109-2628-E-001-001-MY3). This work is part of the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant no. 101008233.

References

Abdulla, P.A., 2012. Regular model checking. STTT 14 (2), 109–118. http://dx.doi.org/10.1007/s10009-011-0216-8.

Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P., 2017. Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017. ACM, pp. 602–617. http://dx.doi.org/10.1145/3062341.3062384.

Abdulla, P.A., Atig, M.F., Chen, Y.-F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P., 2018. Trau: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design. FMCAD, IEEE, pp. 1–5.

Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J., 2014. String constraints for verification. In: International Conference on Computer Aided Verification. Springer, pp. 150–166.

Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J., 2015. Norn: An SMT solver for string constraints. In: International Conference on Computer Aided Verification. Springer, pp. 462–469.

Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P., 2019. Chain-free string constraints. In: ATVA. Springer, pp. 277–293.

Amadini, R., Gange, G., Stuckey, P.J., Tack, G., 2017. A novel approach to string constraint solving. In: Beck, J.C. (Ed.), Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. In: Lecture Notes in Computer Science, vol. 10416, Springer, pp. 3–20. http://dx.doi.org/10.1007/978-3-319-66158-2_1.

Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T., Yu, F., 2018. Parameterized model counting for string and numeric constraints. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 400–410.

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., 2022. CVC5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp. 415–442.

Barceló, P., Figueira, D., Libkin, L., 2013. Graph logics with rational relations. arXiv preprint arXiv:1304.4150.

Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C., 2011. CVC4. In: Gopalakrishnan, G., Qadeer, S. (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. In: Lecture Notes in Computer Science, vol. 6806, Springer, pp. 171–177. http://dx.doi.org/10.1007/978-3-642-22110-1_14.

Berstel, J., 1979. Transductions and Context-Free Languages. In: Teubner Studienbücher : Informatik, vol. 38, Teubner, URL: http://www.worldcat.org/oclc/06364613.

Berzish, M., Ganesh, V., Zheng, Y., 2017. Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017. pp. 55–59. http://dx.doi.org/10.23919/FMCAD.2017.8102241.

Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V., 2021. An SMT solver for regular expressions and linear arithmetic over string length. In: International Conference on Computer Aided Verification. Springer, pp. 289–312.

Bjørner, N., Tillmann, N., Voronkov, A., 2009. Path feasibility analysis for string-manipulating programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009. In: Lecture Notes in Computer Science, vol. 5505, Springer, pp. 307–321. http://dx.doi.org/10.1007/978-3-642-00768-2_27.

Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V., 2018. Stringfuzz: A fuzzer for string solvers. In: International Conference on Computer Aided Verification. Springer, pp. 45–51.

Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T., 2012. Abstract regular (tree) model checking. STTT 14 (2), 167–191. http://dx.doi.org/10.1007/s10009-011-0205-y.

Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T., 2000. Regular model checking. In: Emerson, E.A., Sistla, A.P. (Eds.), Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 403–418.

Büchi, J.R., 1960. Weak second-order arithmetic and finite automata. Math. Log. Q. 6 (1–6), 66–92. http://dx.doi.org/10.1002/malq.19600060105.

Büchi, J.R., Senger, S., 1990. Definability in the existential theory of concatenation and undecidable extensions of this theory. In: The Collected Works of J. Richard Büchi. Springer, pp. 671–683.

Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R., 2006. EXE: Automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. CCS '06, Association for Computing Machinery, pp. 322–335. http://dx.doi.org/10.1145/1180405.1180445.

Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z., 2018. What is decidable about string constraints with the ReplaceAll function. PACMPL 2 (POPL), 3:1–3:29. http://dx.doi.org/10.1145/3158091.

Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z., 2019a. Decision procedures for path feasibility of string-manipulating programs with complex operations. PACMPL 3 (POPL), 49:1–49:30. http://dx.doi.org/10.1145/3290362.

Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z., 2019b. Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. 3 (POPL), 1–30.

Chen, Y., Havlena, V., Lengál, O., Turrini, A., 2020. A symbolic algorithm for the case-split rule in string constraint solving. In: Oliveira, B.C.d.S. (Ed.), Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. In: Lecture Notes in Computer Science, vol. 12470, Springer, pp. 343–363. http://dx.doi.org/10.1007/978-3-030-64437-6_18.

Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B., 2019. On solving word equations using SAT. In: Filiot, E., Jungers, R.M., Potapov, I. (Eds.), Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings. In: Lecture Notes in Computer Science, vol. 11674, Springer, pp. 93–106. http://dx.doi.org/10.1007/978-3-030-30806-3_8.

Demri, S., Lazic, R., 2009. LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. 10 (3), 16:1–16:30. http://dx.doi.org/10.1145/1507244.1507246.

Diekert, V., 2002. Makanin's algorithm. In: Algebraic Combinatorics on Words. In: Encyclopedia of Mathematics and its Applications, Cambridge University Press, pp. 387–442. http://dx.doi.org/10.1017/CBO9781107326019.013.

Durnev, V.G., Zetkina, O.V., 2009. On equations in free semigroups with certain constraints on their solutions. J. Math. Sci. 158 (5), 671–676. http://dx.doi.org/10.1007/s10958-009-9409-z.

Ganesh, V., Berzish, M., 2016. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. arXiv preprint arXiv:1605.09442.

Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M., 2012. Word equations with length constraints: what's decidable? In: Haifa Verification Conference. Springer, pp. 209–226.

Glenn, J., Gasarch, W.I., 1996. Implementing WS1S via finite automata. In: Raymond, D.R., Wood, D., Yu, S. (Eds.), Automata Implementation, First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, August 29-31, 1996, Revised Papers. In: Lecture Notes in Computer Science, vol. 1260, Springer, pp. 50–63. http://dx.doi.org/10.1007/3-540-63174-7_5.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: Directed automated random testing. SIGPLAN Not. 40 (6), 213–223. http://dx.doi.org/10.1145/1064978.1065036.

Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R., 2011. Synthesis of loop-free programs. SIGPLAN Not. 46 (6), 62–73. http://dx.doi.org/10.1145/1993316.1993506.

Gulwani, S., Srivastava, S., Venkatesan, R., 2008. Program analysis as constraint solving. In: PLDI'08.

Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T., 2018. String constraints with concatenation and transducers solved efficiently. Proc. ACM Program. Lang. 2 (POPL), 4:1–4:32. http://dx.doi.org/10.1145/3158092.

Kaminski, M., Francez, N., 1994. Finite-memory automata. Theoret. Comput. Sci. 134 (2), 329–363. http://dx.doi.org/10.1016/0304-3975(94)90242-9.

Karp, R.M., 1972. Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (Eds.), Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA. In: The IBM Research Symposia Series, Plenum Press, New York, pp. 85–103. http://dx.doi.org/10.1007/978-1-4684-2001-2_9.

Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E., 2001. Symbolic model checking with rich assertional languages. Theoret. Comput. Sci. 256 (1–2), 93–112. http://dx.doi.org/10.1016/S0304-3975(00)00103-1.

Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D., 2012. HAMPI: a solver for word equations over strings, regular expressions, and context-free grammars. ACM Trans. Softw. Eng. Methodol. 21 (4), 25:1–25:28. http://dx.doi.org/10.1145/2377656.2377662.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394. http://dx.doi.org/10.1145/360248.360252.

Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J., 2019. Resource-guided program synthesis. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. In: PLDI 2019, Association for Computing Machinery, New York, NY, USA, pp. 253–268. http://dx.doi.org/10.1145/3314221.3314602.

Kosovskii, N.K., 1976. Properties of the solutions of equations in a free semigroup. J. Math. Sci. 6 (4), http://dx.doi.org/10.1007/BF01084074.

Le, Q.L., He, M., 2018. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Ryu, S. (Ed.), Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings. In: Lecture Notes in Computer Science, vol. 11275, Springer, pp. 350–372. http://dx.doi.org/10.1007/978-3-030-02768-1_19.

Levi, F.W., 1944. On semigroups. Bull. Calcutta Math. Soc. 36, 141–146.

Li, G., Ghosh, I., 2013. PASS: String solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (Eds.), Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings. In: Lecture Notes in Computer Science, vol. 8244, Springer, pp. 15–31. http://dx.doi.org/10.1007/978-3-319-03077-7_2.

Liang, T., Reynolds, A., Tinelli, C., Barrett, C.W., Deters, M., 2014. A DPLL(T) theory solver for a theory of strings and regular expressions. In: Computer Aided Verification - 26th International Conference, CAV 2014. In: Lecture Notes in Computer Science, vol. 8559, Springer, pp. 646–662. http://dx.doi.org/10.1007/978-3-319-08867-9_43.

Lin, A.W., Barceló, P., 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: ACM SIGPLAN Notices, Vol. 51. ACM, pp. 123–136.

Lin, A.W., Majumdar, R., 2018. Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018. In: Lecture Notes in Computer Science, vol. 11138, Springer, pp. 352–369. http://dx.doi.org/10.1007/978-3-030-01090-4_21.

Loring, B., Mitchell, D., Kinder, J., 2019. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 425–438.

Makanin, G.S., 1977. The problem of solvability of equations in a free semigroup. Mat. Sb. 145 (2), 147–236.

Matiyasevich, Y.V., 1968. A connection between systems of word and length equations and Hilbert's tenth problem. Zap. Nauchnykh Semin. POMI 8, 132–144.

Matiyasevich, Y., 2008. Computation paradigms in light of Hilbert's tenth problem. In: New Computational Paradigms. Springer, pp. 59–85.

Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V., 2021. Z3str4: A multi-armed string solver. In: Huisman, M., Pasareanu, C.S., Zhan, N. (Eds.), Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. In: Lecture Notes in Computer Science, vol. 13047, Springer, pp. 389–406. http://dx.doi.org/10.1007/978-3-030-90870-6_21.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient SMT solver. 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. In: Lecture Notes in Computer Science, vol. 4963, Springer, pp. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24.

Neider, D., Jansen, N., 2013. Regular model checking using solver technologies and automata learning. In: Brat, G., Rungta, N., Venet, A. (Eds.), NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings. In: Lecture Notes in Computer Science, vol. 7871, Springer, pp. 16–31. http://dx.doi.org/10.1007/978-3-642-38088-4_2.

Nielsen, J., 1917. Die isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Math. Ann. 78 (1), 385–397. http://dx.doi.org/10.1007/BF01457113.

Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C.W., Tinelli, C., 2022. Even faster conflicts and lazier reductions for string solvers. In: Shoham, S., Vizel, Y. (Eds.), Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. In: Lecture Notes in Computer Science, vol. 13372, Springer, pp. 205–226. http://dx.doi.org/10.1007/978-3-031-13188-2_11.

Osera, P.-M., 2019. Constraint-based type-directed program synthesis. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development. In: TyDe 2019, Association for Computing Machinery, New York, NY, USA, pp. 64–76. http://dx.doi.org/10.1145/3331554.3342608.

Pin, J. (Ed.), 2021. Handbook of Automata Theory. European Mathematical Society Publishing House, Zürich, Switzerland, http://dx.doi.org/10.4171/Automata.

Plandowski, W., 1999. Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039). IEEE, pp. 495–500.

Plandowski, W., 2006. An efficient algorithm for solving word equations. In: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing. ACM, pp. 467–476.

Presburger, M., 1929. About the completeness of a certain system of integer arithmetic in which addition is the only operation. In: I Congrès de Mathématiciens des Pays Slaves. pp. 92–101.

Quine, W.V., 1946. Concatenation as a basis for arithmetic. J. Symb. Log. 11 (4), 105–114.

Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C., 2019. High-level abstractions for simplifying extended string constraints in SMT. In: International Conference on Computer Aided Verification. Springer, pp. 23–42.

Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C., 2017. Scaling up DPLL (T) string solvers using context-dependent simplification. In: International Conference on Computer Aided Verification. Springer, pp. 453–474.

Robson, J.M., Diekert, V., 1999. On quadratic word equations. In: Annual Symposium on Theoretical Aspects of Computer Science. Springer, pp. 217–226.

Schulz, K.U., 1990. Makanin's algorithm for word equations-two improvements and a generalization. In: International Workshop on Word Equations and Related Topics. Springer, pp. 85–150.

Scott, J.D., Flener, P., Pearson, J., Schulte, C., 2017. Design and implementation of bounded-length sequence variables. In: Salvagnin, D., Lombardi, M. (Eds.), Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings. In: Lecture Notes in Computer Science, vol. 10335, Springer, pp. 51–67. http://dx.doi.org/10.1007/978-3-319-59776-8_5.

Sen, K., Kalasapur, S., Brutch, T., Gibbs, S., 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2013, Association for Computing Machinery, pp. 488–498. http://dx.doi.org/10.1145/2491411.2491447.

Stanford, C., Veanes, M., Bjørner, N., 2021. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 620–635.

Thatcher, J.W., Wright, J.B., 1968. Generalized finite automata theory with an application to a decision problem of second-order logic. Math. Syst. Theory 2 (1), 57–81. http://dx.doi.org/10.1007/BF01691346.

Trinh, M.-T., Chu, D.-H., Jaffar, J., 2014. S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1232–1243.

Trinh, M.-T., Chu, D.-H., Jaffar, J., 2016. Progressive reasoning over recursively-defined strings. In: International Conference on Computer Aided Verification. Springer, pp. 218–240.

Trinh, M.-T., Chu, D.-H., Jaffar, J., 2020. Inter-theory dependency analysis for SMT string solvers. Proc. ACM Program. Lang. 4 (OOPSLA), 1–27.

Tseitin, G.S., 1983. On the complexity of derivation in propositional calculus. In: Automation of Reasoning. Springer, pp. 466–483.

Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.S., 2012. Symbolic finite state transducers: algorithms and applications. In: Field, J., Hicks, M. (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. ACM, pp. 137–150. http://dx.doi.org/10.1145/2103656.2103674.

Wang, H., Chen, S., Yu, F., Jiang, J.R., 2018. A symbolic model checking approach to the analysis of string and length constraints. In: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. ACM, pp. 623–633. http://dx.doi.org/10.1145/3238147.3238189.

Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R., 2016. String analysis via automata manipulation with logic circuit representation. In: Computer Aided Verification - 28th International Conference, CAV 2016. In: Lecture Notes in Computer Science, vol. 9779, Springer, pp. 241–260. http://dx.doi.org/10.1007/978-3-319-41528-4_13.

Wang, Y., Zhou, M., Jiang, Y., Song, X., Gu, M., Sun, J., 2017. A static analysis tool with optimizations for reachability determination. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 925–930. http://dx.doi.org/10.1109/ASE.2017.8115706.

Wolper, P., Boigelot, B., 1998. Verifying systems with infinite but regular state spaces. In: Hu, A.J., Vardi, M.Y. (Eds.), Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings. In: Lecture Notes in Computer Science, vol. 1427, Springer, pp. 88–97. http://dx.doi.org/10.1007/BFb0028736.

Wolper, P., Boigelot, B., 2000. On the construction of automata from linear arithmetic constraints. In: Graf, S., Schwartzbach, M.I. (Eds.), Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. In: Lecture Notes in Computer Science, vol. 1785, Springer, pp. 1–19. http://dx.doi.org/10.1007/3-540-46419-0_1.

Wolper, P., Boigelot, B., 2002. Representing arithmetic constraints with finite automata: An overview. In: Stuckey, P.J. (Ed.), Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings. In: Lecture Notes in Computer Science, vol. 2401, Springer, pp. 1–19. http://dx.doi.org/10.1007/3-540-45619-8_1.

Yu, F., Alkhalaf, M., Bultan, T., 2010. Stranger: An automata-based string analysis tool for php. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 154–157.

Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H., 2014. Automata-based symbolic string analysis for vulnerability detection. Form. Methods Syst. Des. 44 (1), 44–70. http://dx.doi.org/10.1007/s10703-013-0189-1.

Yu, F., Shueh, C.-Y., Lin, C.-H., Chen, Y.-F., Wang, B.-Y., Bultan, T., 2016. Optimal sanitization synthesis for web application vulnerability repair. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 189–200.

Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X., 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. Form. Methods Syst. Des. 50 (2–3), 249–288.

Yu-Fang Chen is a research fellow at the Institute of Information Science, Academia Sinica. He has more than ten years of experience in the field of formal methods and verification, and has contributed to developing methods such as automata-based verification, algorithmic learning, and various types of decision procedures.

Vojtěch Havlena is a researcher at Faculty of Information Technology, Brno University of Technology, Czech Republic. His research interests include formal methods, automata theory, and logics. He particularly focuses on the development of automata-based techniques for the use in decision procedures of various theories and in security applications.

Ondřej Lengál is an assistant professor at Faculty of Information Technology, Brno University of Technology. His main interests lie in automata theory, logics, and their uses in developing efficient, safe, and secure computer systems. In particular, he works on analysis and verification of programs and techniques for efficient work with automata and their applications

Andrea Turrini received his Master and Ph.D. degree in computer science from University of Verona, Italy, in 2005 and 2009, respectively. He is currently an associate research professor at the Institute of Software, Chinese Academy of Sciences, Beijing. His research interests include the formal analysis of processes involving probability and nondeterminism, the development of efficient operations and transformations on omega regular automata, and the extension of automata-based frameworks to new scenarios.

An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits

YU-FANG CHEN, Academia Sinica, Taiwan
KAI-MIN CHUNG, Academia Sinica, Taiwan
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
JYUN-AO LIN, Academia Sinica, Taiwan
WEI-LUN TSAI, Academia Sinica, Taiwan and National Taiwan University, Taiwan
DI-DE YEN, Max Planck Institute for Software Systems, Germany

We introduce a new paradigm for analysing and finding bugs in quantum circuits. In our approach, the problem is given by a triple $\{P\}\,C\,\{Q\}$ and the question is whether, given a set P of quantum states on the input of a circuit C, the set of quantum states on the output is equal to (or included in) a set Q. While this is not suitable to specify, e.g., functional correctness of a quantum circuit, it is sufficient to detect many bugs in quantum circuits. We propose a technique based on *tree automata* to compactly represent sets of quantum states and develop transformers to implement the semantics of quantum gates over this representation. Our technique computes with an algebraic representation of quantum states, avoiding the inaccuracy of working with floating-point numbers. We implemented the proposed approach in a prototype tool and evaluated its performance against various benchmarks from the literature. The evaluation shows that our approach is quite scalable, e.g., we managed to verify a large circuit with 40 qubits and 141,527 gates, or catch bugs injected into a circuit with 320 qubits and 1,758 gates, where all tools we compared with failed. In addition, our work establishes a connection between quantum program verification and automata, opening new possibilities to exploit the richness of automata theory and automata-based verification in the world of quantum computing.

CCS Concepts: • **Hardware** → **Quantum computation**; • **Theory of computation** → **Tree languages**; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: quantum circuits, tree automata, verification

ACM Reference Format:
Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM Program. Lang.* 7, PLDI, Article 156 (June 2023), 26 pages. https://doi.org/10.1145/3591270

1 INTRODUCTION

The concept of *quantum computing* appeared around 1980 with the promise to solve many problems challenging for classical computers. Quantum algorithms for such problems started appearing later, such as Shor's factoring algorithm [Shor 1994], a solution to the hidden subgroup problem by Ettinger *et al.* [Ettinger et al. 2004], Bernstein-Vazirani's algorithm [Bernstein and Vazirani 1993],

Authors' addresses: Yu-Fang Chen, Academia Sinica, Institute of Information Science, Taiwan, yfc@iis.sinica.edu.tw; Kai-Min Chung, Academia Sinica, Institute of Information Science, Taiwan, kmchung@iis.sinica.edu.tw; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Czech Republic, lengal@fit.vutbr.cz; Jyun-Ao Lin, Academia Sinica, Taiwan, jyalin@gmail.com; Wei-Lun Tsai, Academia Sinica, Institute of Information Science, Taiwan, alan23273850@gmail.com and National Taiwan University, Graduate Institute of Electronics Engineering, Taiwan; Di-De Yen, Max Planck Institute for Software Systems, Germany, bottlebottle13@gmail.com.

2475-1421/2023/6-ART156
https://doi.org/10.1145/3591270

or Grover's search [Grover 1996]. For a long time, no practical implementation of these algorithms has been available due to the missing hardware. Recent years have, however, seen the advent of quantum chips claiming to achieve *quantum supremacy* [Arute et al. 2019], i.e., the ability to solve a problem that a state-of-the-art supercomputer would take thousands of years to solve. As it seems that quantum computers will occupy a prominent role in the future, systems and languages for their programming are in active development (e.g., [Altenkirch and Grattage 2005; Green et al. 2013; Wille et al. 2019]), and efficient quantum algorithms for solutions of real-world problems, such as machine learning [Biamonte et al. 2017; Ciliberto et al. 2018], optimization [Moll et al. 2018], or quantum chemistry [Cao et al. 2019], have started appearing.

The exponential size of the underlying computational space and the probabilistic nature makes it, however, extremely challenging to reason about quantum programs—both for human users and automated analysis tools. Currently, existing automated analysis approaches are mostly unable to handle large-scale circuits [Feng et al. 2017, 2015; Ying 2021; Ying and Feng 2021; Ying et al. 2014], inflexible in checking user-specified properties [Amy 2018; Burgholzer and Wille 2020; Coecke and Duncan 2011; Fagan and Duncan 2019; Green et al. 2013; Niemann et al. 2016; Pednault et al. 2017; Samoladas 2008; Tsai et al. 2021; Viamontes et al. 2009; Wecker and Svore 2014; Zulehner et al. 2019; Zulehner and Wille 2019], or imprecise and unable to catch bugs [Perdrix 2008; Yu and Palsberg 2021]. Scalable and flexible automated analysis tools for quantum circuits are indeed missing.

In this paper, we propose a new paradigm for analysing and finding bugs in quantum circuits. In our approach, the problem is given by a triple $\{P\}\, C\, \{Q\}$, where C is a quantum circuit and P and Q are sets of quantum states. The verification question that we address is whether the set of output quantum states obtained by running C on all states from P is equal to (or included in) the set Q. While this kind of property is not suitable to specify, e.g., functional correctness of a quantum circuit, it is sufficient to obtain a lot of useful information about a quantum circuit, such as finding constants (will a circuit evaluate to the same quantum state for all inputs in P) or detecting bugs.

We create a framework for analysing the considered class of properties based on *(finite) tree automata* (TAs) [Comon et al. 2008]. Languages of TAs are set of trees; in our case, we consider TAs whose languages contain full binary trees with the height being the number of qubits in the circuit. Each branch (a path from a root to a leaf) in such a tree corresponds to one *computational basis state* (e.g., $|0000\rangle$ or $|0101\rangle$ for a four-qubit circuit), and the corresponding leaf represents the *complex amplitude* of the state (we use an algebraic encoding of complex numbers by tuples of integers to have a precise representation and avoid possible inaccuracies when dealing with floating-point numbers[1]; this encoding is sufficient for a wide variety of quantum gates, including the Clifford+T universal set [Boykin et al. 2000]). Sets of such trees can be in many cases encoded compactly using TAs, e.g., storing the output of Bernstein-Vazirani's algorithm [Bernstein and Vazirani 1993] over n qubits requires a vector of 2^n complex numbers, but can be encoded by a linear-sized TA. For each quantum gate, we construct a transformation that converts the input states TA to a TA representing the gate's output states, in a similar way as classical program transformations are represented in [D'Antoni et al. 2015]. Testing equivalence and inclusion between the TA representing the set of outputs of a circuit and the postcondition Q (from $\{P\}\, C\, \{Q\}$) can then be done by standard TA language inclusion/equivalence testing algorithms [Abdulla et al. 2008, 2007; Comon et al. 2008; Lengál et al. 2012]. If the test fails, the framework generates a witness for diagnosis.

One application of our framework is as a quick *underapproximation of a quantum circuit non-equivalence test*. Our approach can switch to a lightweight specification when equivalence checkers fail due to insufficient resources and still find bugs in the design. Quantum circuit (non-)equivalence testing is an essential part of the quantum computing toolkit. Its prominent use is in verifying

[1]Integer numbers of an arbitrary precision can be handled, e.g., by the popular `GMP` [GMP 2022] package.

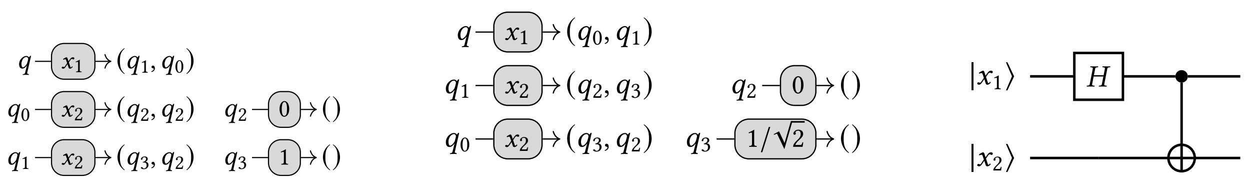
results of circuit optimization, which is a necessary part of quantum circuit compilation in order to achieve the expected fidelity of quantum algorithms running on real-world quantum computers, which are heavily affected by noise and decoherence [Amy 2019; Hattori and Yamashita 2018; Hietala et al. 2019; Itoko et al. 2020; Moll et al. 2018; Nam et al. 2018; Peham et al. 2022; Soeken et al. 2010; Xu et al. 2022b]. Already in the world of classical programs, optimizer bugs are being found on a regular basis in compilers used daily by tens of thousands of programmers (see, e.g., [Livinskii et al. 2020]). In the world of quantum, optimization is much harder than in the classical setting, with many opportunities to introduce subtle and hard-to-discover bugs into the optimized circuits. It is therefore essential to be able to check that an output of an optimizer is functionally equivalent to its input. Moreover, global optimization techniques, such as genetic algorithms [Massey et al. 2005; Spector 2006], may use (somehow quantified) circuit (non-)equivalence as the fitness function.

Testing quantum circuit (non-)equivalence is, however, a challenging task (QMA-complete [Janzing et al. 2005]). Due to its complexity, approaches that can quickly establish circuit non-equivalence are highly desirable to be used, e.g., as a preliminary check before a more heavy-weight procedure, such as [Burgholzer and Wille 2020; Peham et al. 2022; Viamontes et al. 2007; Wei et al. 2022; Yamashita and Markov 2010], is used. One currently employed fast non-equivalence check is to use random stimuli generation [Burgholzer et al. 2021]. Finding subtle bugs by random testing is, however, challenging with no guarantees due to the immense (in general uncountable) underlying state space.

Our approach can be used as follows: we start with a TA encoding the set of possible input states (created by the user or automatically) and run our analysis of the circuit over it, obtaining a TA $\mathcal{A}$ representing the set of all outputs. Then, we take the optimized circuit, run it over the same TA with inputs and obtain a TA $\mathcal{A}'$. Finally, we check whether $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. If the equality does not hold, we can conclude that the circuits are not functionally equivalent (if the equality holds, there can, however, still be some bug that does not manifest in the set of output states).

We implemented our technique in a prototype called AutoQ and evaluated it over a wide range of quantum circuits, including some prominent quantum algorithms, randomly generated circuits, reversible circuits from RevLib [Wille et al. 2008], and benchmarks from the tool Feynman [Amy 2018]. The results show that our approach is quite scalable. We did not find any tool with the same functionality with ours and hence pick the closest state-of-the art tools: a circuit simulator SliQSim [Tsai et al. 2021] and circuit equivalence checkers Feynman [Amy 2018] (based on path-sum) and Qcec [Burgholzer and Wille 2020] (combining ZX-calculus, decision diagrams, and random stimuli generation), as the baseline tools to compare with. In the first experiments, we evaluated AutoQ's capability in verification against pre- and post-conditions. We managed to verify the functional correctness (w.r.t. one input state) of a circuit implementing Grover's search algorithm with 40 qubits and 141,527 gates. We then evaluated AutoQ on circuits with injected bugs. The results confirm our claim—AutoQ was able to find injected bugs in various huge-scale circuits, including one with 320 qubits and 1,758 gates, which the other tools failed to find.

In addition to the practical utility, our work also bridges the gap between quantum and classical verification, particularly automata-based approaches such as *regular (tree) model checking* [Bouajjani et al. 2012, 2000; Neider and Jansen 2013] or string manipulation verification [Yu et al. 2008, 2011]. As far as we know, our approach to verification of quantum circuits is the first based on automata. The enabling techniques and concepts involved in this work are, e.g., the use of TAs to represent sets of quantum states and express the pre- and post-conditions, the compactness of the TA structure enabling efficient gate operations, and our TA transformation algorithms enabling the execution of quantum gates over TAs. We believe that the connection of automata theory with the quantum world we establish can start new fruitful collaborations between the two rich fields.



(a) The TA of $|00\rangle$.

(b) The TA of $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

(c) The EPR circuit

Fig. 1. Constructing the Bell state

Overview: We use a concrete example to demonstrate how to use our approach. Assume that we want to design a circuit constructing the Bell state, i.e., a 2-qubit circuit converting a basis state $|00\rangle$ to a maximally entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. We first prepare TAs corresponding to the precondition (Fig. 1a) and postcondition (Fig. 1b). Both TAs use q as the root state and accept only one tree. One can see the correspondence between quantum states and TAs by traversing their structure. The precise definition will be given in Section 2 and Section 3. Our approach will then use the transformers from Sections 4 to 6 to construct a TA $\mathcal{A}$ recognizing the quantum states after executing the EPR circuit (Fig. 1c) from the precondition TA (Fig. 1a). We will then use TA language inclusion/equivalence tool VATA [Lengál et al. 2012] to check $\mathcal{A}$ against the postcondition TA. If the circuit is buggy, our approach will return a witness quantum state that is reachable from the precondition, but not allowed by the postcondition. From our experience of preparing benchmark examples, in many cases, this approach helps us finding out bugs from incorrect designs.

2 PRELIMINARIES

We assume basic knowledge of linear algebra and quantum circuits. Below, we only give a short overview and fix notation; see, e.g., the textbook [Nielsen and Chuang 2011] for more details.

By default, we work with vectors and matrices over complex numbers $\mathbb{C}$. In particular, we use $\mathbb{C}^{m\times n}$ to denote the set of all $m \times n$ complex matrices. Given a $k \times \ell$ matrix (a_{xy}), its *transpose* is the $\ell \times k$ matrix $(a_{xy})^T = (a_{yx})$ obtained by flipping (a_{xy}) over its diagonal. A $1 \times k$ matrix is called a *row vector* and a $k \times 1$ matrix is called a column vector. To save vertical space, we often write a column vector v using its row transpose v^T. We use I to denote the *identity* matrix of any dimension (which should be clear from the context). The *conjugate* of a complex number $a+bi$ is the complex number $a - bi$, and the *conjugate transpose* of a matrix $A = (a_{xy})$ is the matrix $A^\dagger = (c_{yx})$ where c_{yx} is the conjugate of a_{yx}. For instance, $\begin{pmatrix} 1+i & 2-2i & 3 \\ 4-7i & 0 & 0 \end{pmatrix}^\dagger = \begin{pmatrix} 1-i & 4+7i \\ 2+2i & 0 \\ 3 & 0 \end{pmatrix}$. The *inverse* of a matrix A is denoted as A^{-1}. A square matrix A is *unitary* if $A^\dagger = A^{-1}$. The *Kronecker product* of $A = (a_{xy}) \in \mathbb{C}^{k\times\ell}$ and $B \in \mathbb{C}^{m\times n}$ is the $km \times \ell n$ matrix $A \otimes B = (a_{xy}B)$, for instance,

$$\begin{pmatrix} 1+i & 3 \\ 4-7i & 0 \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} = \begin{pmatrix} (1+i)\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} & 3\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} \\ (4-7i)\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} & 0\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{2}+\frac{1}{2}i & 1+i & \frac{3}{2} & 3 \\ -\frac{1}{2}-\frac{1}{2}i & 0 & -\frac{3}{2} & 0 \\ 2-\frac{7}{2}i & 4-7i & 0 & 0 \\ -2+\frac{7}{2}i & 0 & 0 & 0 \end{pmatrix}. \tag{1}$$

2.1 Quantum Circuits

Quantum states. In a quantum system with n qubits, the qubits can be entangled, and its *quantum state* can be a quantum superposition of *computational basis states* $\{|j\rangle \mid j \in \{0,1\}^n\}$. For instance, given a system with three qubits x_1, x_2, and x_3, the computational basis state $|011\rangle$ denotes a state where qubit x_1 is set to 0 and qubits x_2 and x_3 are set to 1. The superposition is then denoted in the

Dirac notation as a formal sum $\sum_{j\in\{0,1\}^n} a_j \cdot |j\rangle$, where $a_0, a_1, \ldots, a_{2^n-1} \in \mathbb{C}$ are *complex amplitudes*[2] satisfying the property that $\sum_{j\in\{0,1\}^n} |a_j|^2 = 1$. Intuitively, $|a_j|^2$ is the probability that when we measure the state in the computational basis, we obtain the state $|j\rangle$; these probabilities need to sum up to 1 for all computational basis states. We note that the quantum state can alternatively be represented by a 2^n-dimensional column vector[3] $(a_0, \ldots, a_{2^n-1})^T$ or by a function $T\colon \{0,1\}^n \to \mathbb{C}$, where $T(j) = a_j$ for all $j \in \{0,1\}^n$. In the following, we will work mainly with the function representation, which we will see as a mapping from the domain of assignments to Boolean variables (corresponding to qubits) to $\mathbb{C}$. For instance, the quantum state $\frac{1}{\sqrt{2}} \cdot |00\rangle + \frac{1}{\sqrt{2}} \cdot |01\rangle$ can be represented by the vector $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0)^T$ or the function $T = \{00 \mapsto \frac{1}{\sqrt{2}}, 01 \mapsto \frac{1}{\sqrt{2}}, 10 \mapsto 0, 11 \mapsto 0\}$.

Quantum gates. Operations in quantum circuits are represented using quantum gates. A k-qubit *quantum gate* (i.e., a quantum gate with k inputs and k outputs) can be described using a $2^k \times 2^k$ unitary matrix. When computing the effect of a k-qubit quantum gate U on the qubits $x_\ell, \ldots, x_{\ell+k-1}$ of an n-qubit quantum state represented using a 2^n-dimensional vector v, we proceed as follows. First, we compute an auxiliary matrix $U' = I_{n-(\ell+k-1)} \otimes U \otimes I_{\ell-1}$ where I_j denotes the 2^j-dimensional identity matrix. Note that if U is unitary, then U' is also unitary. Then, the new quantum state is computed as $v' = U' \times v$. For instance, let $n = 2$ and U be the Pauli-X gate applied to the qubit x_1.

$$X' = X \otimes I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad v' = X' \times v = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{pmatrix} = \begin{pmatrix} c_{10} \\ c_{11} \\ c_{00} \\ c_{01} \end{pmatrix} \tag{2}$$

Representation of complex numbers. In order to achieve accuracy with no loss of precision, in this paper, when working with $\mathbb{C}$, we consider only a subset of complex numbers that can be expressed by the following algebraic encoding proposed in [Zulehner and Wille 2019] (and also used in [Tsai et al. 2021]):

$$\left(\frac{1}{\sqrt{2}}\right)^k (a + b\omega + c\omega^2 + d\omega^3), \tag{3}$$

where $a, b, c, d, k \in \mathbb{Z}$ and $\omega = e^{\frac{i\pi}{4}}$, the unit vector that makes an angle of 45° with the positive real axis in the complex plane). A complex number is then represented by a five-tuple (a, b, c, d, k). Although the considered set of complex numbers is only a small subset of $\mathbb{C}$ (it is countable, while the set $\mathbb{C}$ is uncountable), the subset is already sufficient to describe a set of quantum gates that can implement universal quantum computation (cf. Section 4 for more details). The algebraic representation also allows efficient encoding of some operations. For example, because $\omega^4 = -1$, the multiplication of (a, b, c, d, k) by ω can be carried out by a simple right circular shift of the first four entries and then taking the opposite number for the first entry, namely $(-d, a, b, c, k)$, which represents the complex number $(\frac{1}{\sqrt{2}})^k(-d + a\omega + b\omega^2 + c\omega^3)$. In the rest of the paper, we use $\mathbf{0}$ and $\mathbf{1}$ to denote the tuples for zero and one, i.e., $(0,0,0,0,0)$ and $(1,0,0,0,0)$, respectively. Using such an encoding, we represent quantum states by functions of the form $T\colon \{0,1\}^n \to \mathbb{Z}^5$.

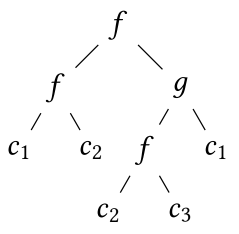
Qubit Measurement. After executing a quantum circuit, one can measure the final quantum state in the computational basis. The probability that the qubit x_j of a quantum state $\sum_{i\in\{0,1\}^n} a_i \cdot |i\rangle$ is measured as the basis state $|0\rangle$ can be computed from the amplitude: $Prob[x_j = |0\rangle] = \sum_{i\in\{0,1\}^{n-j}\times\{0\}\times\{0,1\}^{j-1}} |a_i|^2$. When x_j collapses to $|0\rangle$ after the measurement, amplitudes of states with $x_j = |1\rangle$ become 0 and amplitudes of states with $x_j = |0\rangle$ are normalized using $\frac{1}{\sqrt{Prob[x_j=|0\rangle]}}$.

[2]We abuse notation and sometimes identify a binary string with its (unsigned) integer value in the *most significant bit first* (MSBF) encoding, e.g., the string 0101 with the number 5.

[3]Observe that in order to satisfy the requirement for the amplitudes of quantum states, it must be a *unit* vector.

2.2 Tree Automata

Binary Trees. We use a ranked alphabet Σ with binary symbols $f, g, \ldots$ and constant symbols $c_1, c_2, \ldots$. A *binary tree* is a *ground term* over Σ. For instance, $T = f(f(c_1, c_2), g(f(c_2, c_3), c_1))$, shown in the right, represents a binary tree. The set of *nodes* of a binary tree T, denoted as N_T, is defined inductively as a set of words over $\{0, 1\}$ such that for every constant symbol c, we define $N_c = \{\epsilon\}$, and for every binary symbol f, we define $N_{f(T_0,T_1)} = \{\epsilon\} \cup \{a.w \mid a \in \{0, 1\} \wedge w \in N_{T_a}\}$, where ϵ is the empty word and '.' is concatenation. Each binary tree T is associated with a labeling function $L_T\colon \{0, 1\}^* \longrightarrow \Sigma$, which maps a node in T to its label in Σ. A tree is *single-valued* if it contains only one constant symbol.

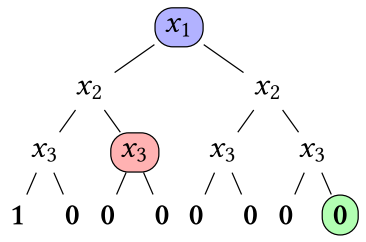


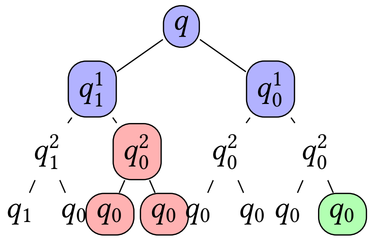
Tree Automata. We focus on tree automata on binary trees and refer the interested reader to [Comon et al. 2008] for a general definition. A *(nondeterministic finite) tree automaton* (TA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$ where Q is a finite set of *states*, Σ is a ranked alphabet, $\mathcal{R} \subseteq Q$ is the set of *root states*, and $\Delta = \Delta_i \cup \Delta_l$ is a set of tree transitions consisting of the set Δ_i of *internal transitions* of the form $q \xrightarrow{\boxed{f}} (q_0, q_1)$ (for a binary symbol f) and the set Δ_l of *leaf transitions* of the form $q \xrightarrow{\boxed{c}} ()$ (for a constant symbol c), for $q, q_0, q_1 \in Q$. W.l.o.g., to simplify our correctness proof, we assume every leaf transition of TAs has a unique parent state, namely, for any two leaf transitions $q \xrightarrow{\boxed{c}} (), q' \xrightarrow{\boxed{c'}} () \in \Delta$, it holds that $c \neq c' \implies q \neq q'$. We can conveniently describe TAs by providing only the set of root states $\mathcal{R}$ and the set of transitions Δ. The alphabet and states are implicitly defined as those that appear in Δ. For example, $\Delta = \{q \xrightarrow{\boxed{x_1}} (q_1, q_0), q \xrightarrow{\boxed{x_1}} (q_0, q_1), q_0 \xrightarrow{\boxed{\mathbf{0}}} (), q_1 \xrightarrow{\boxed{\mathbf{1}}} ()\}$ implies that $\Sigma = \{x_1, \mathbf{0}, \mathbf{1}\}$ and $Q = \{q, q_0, q_1\}$.

Run and Language. A *run* of $\mathcal{A}$ on a tree T is another tree ρ labeled with Q such that (i) T and ρ have the same set of nodes, i.e., $N_T = N_\rho$, (ii) for all leaf nodes $u \in N_T$, we have $L_\rho(u) \xrightarrow{\boxed{L_T(u)}} () \in \Delta$, and (iii) for all non-leaf nodes $v \in N_T$, we have $L_\rho(u) \xrightarrow{\boxed{L_T(u)}} (L_\rho(0.u), L_\rho(1.u)) \in \Delta$. The run ρ is *accepting* if $L_\rho(\epsilon) \in \mathcal{R}$. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \{T \mid \text{there exists an accepting run of } \mathcal{A} \text{ over } T\}$. A TA is (top-down) *deterministic* if it has at most one root state and for any of its transitions $q \xrightarrow{\boxed{x}} (q_l, q_r)$ and $q \xrightarrow{\boxed{x}} (q_l', q_r')$ it holds that $q_l = q_l'$ and $q_r = q_r'$. Any tree from the language of a deterministic TA has a unique run in the TA.

Example 2.1 (Accepted tree and its run). Assume a TA $\mathcal{A}_3$ with q as its single root state and the following transitions:

$$\begin{array}{llll}
q \xrightarrow{\boxed{x_1}} (q_0^1, q_1^1) & q_1^1 \xrightarrow{\boxed{x_2}} (q_0^2, q_1^2) & q_1^2 \xrightarrow{\boxed{x_3}} (q_0, q_1) & q_0 \xrightarrow{\boxed{\mathbf{0}}} () \\
q \xrightarrow{\boxed{x_1}} (q_1^1, q_0^1) & q_1^1 \xrightarrow{\boxed{x_2}} (q_1^2, q_0^2) & q_1^2 \xrightarrow{\boxed{x_3}} (q_1, q_0) & q_1 \xrightarrow{\boxed{\mathbf{1}}} () \\
 & q_0^1 \xrightarrow{\boxed{x_2}} (q_0^2, q_0^2) & q_0^2 \xrightarrow{\boxed{x_3}} (q_0, q_0) &
\end{array}$$





Among others, $\mathcal{A}_3$ accepts the above tree (in the left) with the run (in the right). Observe that all tree nodes satisfy the requirement of a valid run. E.g., the node $\boxed{111}$ corresponds to the transition $q_0 \xrightarrow{\boxed{\mathbf{0}}} ()$, $\boxed{01}$ to $q_0^2 \xrightarrow{\boxed{x_3}} (q_0, q_0)$, and $\boxed{\epsilon}$ to $q \xrightarrow{\boxed{x_1}} (q_1^1, q_0^1)$, etc.

In $\mathcal{A}_3$, we use states named q_0^n to denote only subtrees with all zeros ($\mathbf{0}$) in leaves that can be generated from here, and states named q_1^n to denote only subtrees with a single $\mathbf{1}$ in the leaves that can be generated from it. Intuitively, the TA accepts all trees of the height three with exactly one $\mathbf{1}$ leaf and all other leaves $\mathbf{0}$ (in our encoding of quantum states, this might correspond to saying that $\mathcal{A}_3$ encodes an arbitrary computational basis state of a three-qubit system). □

3 ENCODING SETS OF QUANTUM STATES WITH TREE AUTOMATA

Observe that we can use (full) binary trees to encode functions $\{0,1\}^n \to \mathbb{Z}^5$, i.e., the function representation of quantum states. For instance, the tree

$$x_1(x_2(x_3(\mathbf{1},\mathbf{0}),x_3(\mathbf{0},\mathbf{0})),x_2(x_3(\mathbf{0},\mathbf{0}),x_3(\mathbf{0},\mathbf{0}))) \tag{4}$$

encodes the function T where $T(000) = \mathbf{1}$ and $T(i) = \mathbf{0}$ for all $i \in \{0,1\}^3 \setminus \{000\}$. Since TAs can concisely represent sets of binary trees, they can be used to encode sets of quantum states.

Example 3.1 (Concise representation of sets of quantum states by TAs). Here we consider the set of n-qubit quantum states $Q_n = \{|i\rangle \mid i \in \{0,1\}^n\}$, i.e., the set of all basis states. Note that $|Q_n| = 2^n$, which is exponential. Representing all possible basis states naively would require storing 2^{2^n} complex numbers. TAs can, however, represent such a set much more efficiently.

For the case when $n = 3$, the set Q_3 can be represented by the TA $\mathcal{A}_3$ from Example 2.1 with $3n + 1$ transitions (i.e., linear-sized). The TA $\mathcal{A}_3$ can be generalized to encode the set of all n-qubit states $Q_n = \{|i\rangle \mid i \in \{0,1\}^n\}$ for each $n \in \mathbb{N}$ by setting the transitions to

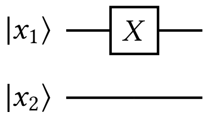
$$\begin{array}{lllll}
q - \boxed{x_1} \rightarrow (q_0^1, q_1^1) & q_1^1 - \boxed{x_2} \rightarrow (q_0^2, q_1^2) & \dots & q_1^{n-1} - \boxed{x_n} \rightarrow (q_0, q_1) & q_0 - \boxed{\mathbf{0}} \rightarrow () \\
q - \boxed{x_1} \rightarrow (q_1^1, q_0^1) & q_1^1 - \boxed{x_2} \rightarrow (q_1^2, q_0^2) & \dots & q_1^{n-1} - \boxed{x_n} \rightarrow (q_1, q_0) & q_1 - \boxed{\mathbf{1}} \rightarrow () \\
 & q_0^1 - \boxed{x_2} \rightarrow (q_0^2, q_0^2) & \dots & q_0^{n-1} - \boxed{x_n} \rightarrow (q_0, q_0) &
\end{array}$$

We denote the resulting TA by $\mathcal{A}_n$. Notice that although Q_n has 2^n quantum states, $\mathcal{A}_n$ has only $2n + 1$ states and $3n + 1$ transitions. □

Formally a TA $\mathcal{A}$ recognizing a set of quantum states is a tuple $\langle Q, \Sigma, \Delta, \mathcal{R} \rangle$, whose alphabet Σ can be partitioned into two classes of symbols: binary symbols $x_1, \dots, x_n$ and a finite set of leaf symbols $\Sigma_c \subseteq \mathbb{Z}^5$ representing all possible amplitudes of quantum states in terms of computational bases. By slightly abusing the notation, for a full binary tree $T \in \mathcal{L}(\mathcal{A})$, we also use T to denote the function $\{0,1\}^n \to \mathbb{Z}^5$ that maps a computational basis to the corresponding amplitude of T's quantum state. The two meanings of T are used interchangeably throughout the paper.

Remark. Note that TAs allow representation of *infinite* languages, yet we only use them for *finite* sets, which might seem like the model is overly expressive. We, however, stick to TAs for the following two reasons: (i) there is an existing rich toolbox for TA manipulation and minimization, e.g., [Abdulla et al. 2008, 2007; Comon et al. 2008; Lengál et al. 2012], and (ii) we want to have a robust formal model for extending our framework to parameterized verification, i.e., proving that an n-qubit algorithm is correct for any n, which will require us to deal with infinite languages (c.f., the framework of *regular tree model checking* [Abdulla et al. 2002; Bouajjani et al. 2012]).

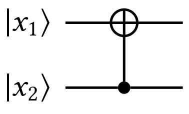
Moreover, we chose *full* binary trees as the representation of quantum states. We thought about using a more compact structure, e.g., allowing jump over a transition with common left and right children (similar to ROBDD's elimination of a node with isomorphic subtrees [Bryant 1986]). We decided against that because TAs already allow an efficient representation of common children via a transition to the same left and right states, e.g., $q - \boxed{x} \rightarrow (q', q')$. The benefit of using a more compact tree representation is thus limited. Using a more efficient data structure would also make



(a) X gate applied to qubit x_1

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(b) Matrix of the X gate



(c) $CNOT_1^2$ gate with target qubit x_1 and control qubit x_2

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(d) Matrix of the $CNOT$ gate

Fig. 2. Applications of X and $CNOT$ gates and their matrices

the algorithms in the following sections harder to understand. We therefore leave the investigation of designing a more efficient data structure to our future work.

4 SYMBOLIC REPRESENTATION OF QUANTUM GATES

With TAs used to concisely represent sets of quantum states, the next task is to capture the effects of applying quantum gates on this representation. When quantum states are represented as vectors, gates are represented as matrices and gate operations are matrix multiplications. When states are represented as binary trees, we need a new representation for quantum gates and their operations. Inspired by the work of [Tsai et al. 2021], we introduce *symbolic update formulae*, which are formulae that describe how a gate transforms a tree representing a quantum state. Later, we will lift the tree update operation to a set of trees encoded in a TA.

We use the algebraic representation of quantum states from Eq. (3) also for their symbolic handling. For instance, consider a system with qubits x_1, x_2 and its state

$$T = c_{00} \cdot |00\rangle + c_{01} \cdot |01\rangle + c_{10} \cdot |10\rangle + c_{11} \cdot |11\rangle \tag{5}$$

for $c_{00}, c_{01}, c_{10}, c_{11} \in \mathbb{Z}^5$, four complex numbers represented in the algebraic way. The result of applying the X gate (the quantum version of the NOT gate) on qubit x_1 (cf. Fig. 2a) is $(c_{10}, c_{11}, c_{00}, c_{01})^T$ (cf. Eq. (2)). Intuitively, we observe that the effect of the gate is a permutation of the computational basis states that swaps the amplitudes of states where the x_1's value is 1 with states where the x_1's value is 0 (and the values of qubits other than x_1 stay the same). Concretely, it swaps the amplitudes of the pairs $(|\mathbf{0}0\rangle, |\mathbf{1}0\rangle)$ and $(|\mathbf{0}1\rangle, |\mathbf{1}1\rangle)$ to obtain the quantum state

$$X(T) = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle\,. \tag{6}$$

Instead of executing the quantum gate by performing a matrix-vector multiplication, we will capture its semantics *symbolically* by directly manipulating the tree function $T\colon \{0,1\}^n \to \mathbb{Z}^5$. For this, we will use the following operators on T, parameterized by a qubit x_t (t for "target"):

$$T_{x_t}(b_n \dots b_t \dots b_1) = T(b_n \dots 1 \dots b_1) \qquad B_{x_t}(b_n \dots b_t \dots b_1) = b_t$$

$$T_{\overline{x_t}}(b_n \dots b_t \dots b_1) = T(b_n \dots 0 \dots b_1) \qquad B_{\overline{x_t}}(b_n \dots b_t \dots b_1) = \overline{b_t}.$$

(Projection) (Restriction)

In the previous, $\overline{b_t}$ denotes the complement of the bit b_t (i.e., $\overline{0} = 1$ and $\overline{1} = 0$). Intuitively, T_{x_t} and $T_{\overline{x_t}}$ fix the value of qubit x_t to be 1 and 0 respectively. On the other hand, B_{x_t} and $B_{\overline{x_t}}$ just take the value of qubit x_t (or its negation) in the computational basis state.

Equipped with the operators, we can now proceed to express the semantics of X symbolically. Let us first look at the first two summands on the right-hand side of Eq. (6): $T^0 = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle$. These summands can be obtained by manipulating the input function T in the following way:

$$T^0 = B_{\overline{x_1}} \cdot T_{x_1}. \tag{7}$$

Table 1. Symbolic update formulae for the considered quantum gates; x_c and x'_c denote control bits (if they exist), and x_t denotes the target bit.

Gate	Update
X_t	$B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}$
Y_t	$\omega^2 \cdot (B_{x_t} \cdot T_{\overline{x_t}} - B_{\overline{x_t}} \cdot T_{x_t})$
Z_t	$B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T$
H_t	$(T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t} - B_{x_t} \cdot T)/\sqrt{2}$
S_t	$B_{\overline{x_t}} \cdot T + \omega^2 \cdot B_{x_t} \cdot T$
T_t	$B_{\overline{x_t}} \cdot T + \omega \cdot B_{x_t} \cdot T$
$\mathrm{Rx}(\frac{\pi}{2})_t$	$(T - \omega^2 \cdot (B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}))/\sqrt{2}$
$\mathrm{Ry}(\frac{\pi}{2})_t$	$(T_{\overline{x_t}} + B_{x_t} \cdot T - B_{\overline{x_t}} \cdot T_{x_t})/\sqrt{2}$
CNOT_t^c	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})$
CZ_t^c	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T)$
$\mathrm{Toffoli}_t^{c,c'}$	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_{c'}}} \cdot T + B_{x_{c'}} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}))$

Here, $T^0 = B_{\overline{x_1}} \cdot T_{x_1}$ is a shorthand for $T^0(b_1 \ldots b_n) = B_{\overline{x_1}}(b_1 \ldots b_n) \cdot T_{x_1}(b_1 \ldots b_n)$. When we view T as a tree, the operation T_{x_1} essentially copies the right subtree of every x_1-node to its left subtree, and $B_{\overline{x_1}} \cdot T_{x_1}$ makes all leaves in every right subtree of T_{x_1}'s x_1-node zero. This would give us

$$T^0 \quad = \quad c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + 0 \cdot |10\rangle + 0 \cdot |11\rangle \quad = \quad c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle . \tag{8}$$

On the other hand, the last two summands in the right-hand side of Eq. (6), i.e., $T^1 = c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle$, could be obtained by manipulating T as follows:

$$T^1 = B_{x_1} \cdot T_{\overline{x_1}}. \tag{9}$$

The tree view of $B_{x_1} \cdot T_{\overline{x_1}}$ is symmetric to $B_{\overline{x_1}} \cdot T_{x_1}$, which would give us the following state:

$$T^1 \quad = \quad 0 \cdot |00\rangle + 0 \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle \quad = \quad c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle . \tag{10}$$

Finally, by summing T^0 and T^1, we obtain Eq. (6): $T^0 + T^1 = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle$. That is, the semantics of the X gate could be expressed using the following symbolic formula:

$$X_1(T) = B_{\overline{x_1}} \cdot T_{x_1} + B_{x_1} \cdot T_{\overline{x_1}}. \tag{11}$$

Observe that the sum effectively swaps the left and right subtrees of each x_1-node.

For multi-qubit gates, the update formulae get more complicated, since they involve more than one qubit. Consider, e.g., the "controlled-NOT" gate $CNOT_t^c$ (see Fig. 2c for the graphical representation and Fig. 2d for its semantics). The $CNOT_t^c$ gate uses x_t and x_c as the target and control qubit respectively. Intuitively, it "flips" the target qubit's value when the control qubit's value is 1 and keeps the original value if it is 0. Similarly, as for the X gate, we can deduce a symbolic formula for the update done by a $CNOT$ gate:

$$CNOT_t^c(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}). \tag{12}$$

The sum consists of the following two summands:

- The summand $B_{\overline{x_c}} \cdot T$ says that when the control qubit is 0, x_t and x_c stay the same.
- The summand $B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})$ handles the case when x_c is 1. In such a case, we apply the X gate on x_t (observe that the inner term is the update formula of X_t in Eq. (11)).

One can obtain symbolic update formulae for other quantum gates in a similar way. In Table 1 we give the formulae for the gates supported by our framework (see [Chen et al. 2023b] for their usual definition using matrices).

For a gate G, we use the superscripts c and c' to denote that x_c and x'_c are the gate's control qubits (if they exist) and the subscript t to denote that x_t is the target bit (e.g., $\mathrm{G}_t^{c,c'}$). We note that the supported set of gates is much larger than is required to achieve (approximate) universal quantum computation (for which it suffices to have, e.g., (i) Clifford gates (H, S, and $CNOT$) and T (see [Boykin et al. 2000]) or (ii) Toffoli and H (see [Aharonov 2003])).

Theorem 4.1. *The symbolic update formulae in Table 1 are correct (w.r.t. the standard semantics of quantum gates, cf. [Nielsen and Chuang 2011]).*

A note on expressivity. The expressivity of our framework is affected by the following factors:

(1) *Algebraic complex number representation* (a, b, c, d, k): This representation can arbitrarily closely approximate any complex number: First, note that $\omega = \cos 45° + i \sin 45° = \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}}$ and when $b = d = 0$, we have $(a, 0, c, 0, k) = \frac{1}{\sqrt{2}^k}(a + c\omega^2) = \frac{a}{\sqrt{2}^k} + \frac{ci}{\sqrt{2}^k}$. Then any complex number can be approximated arbitrarily closely by picking suitable a, c, and k.
(2) *Supported quantum gates*: We covered all standard quantum gates supported in modern quantum computers except parameterized rotation gate. From Solovay-Kitaev theorem [Dawson and Nielsen 2006], gates performing rotations by $\frac{\pi}{2^k}$ can be approximated with an error rate ϵ with $O(\log^{3.97}(\frac{1}{\epsilon}))$-many gates that we support.
(3) *Tree automata structure*: We use non-deterministic transitions of tree automata to represent a set of trees compactly. Nevertheless, we can currently encode only a finite set of states, so encoding, e.g., all quantum states that satisfy $| |10\rangle | = | |01\rangle |$ is future work.

In the next two sections, we discuss how to lift the tree update operation to a set of trees encoded in a TA. Our framework allows different instantiations. We will introduce two in this paper, namely the (i) *permutation-based* (Section 5) and (ii) *composition-based* (Section 6) approach. The former is simple, efficient, and works for all but the H_t, $\mathrm{Rx}(\frac{\pi}{2})_t$, and $\mathrm{Ry}(\frac{\pi}{2})_t$ gates from Table 1 (those whose effect is a permutation of tree leaves, i.e., for gates whose matrix contains only one non-zero element in each row, potentially with a constant scaling of amplitude), while the latter supports all gates in the table but is less efficient. The two approaches are compatible with each other, so one can, e.g., choose to use the permutation-based approach by default and for unsupported gates fall back on the composition-based approach.

5 PERMUTATION-BASED ENCODING OF QUANTUM GATES

Let us first look at the simplest gate $\mathrm{X}_t(T) = B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}$. Recall that in Section 4, we showed that the formula essentially swaps the left and right subtrees of each x_t-labeled node. For a TA $\mathcal{A}$, we can capture the effect of applying X_t to all states in $\mathcal{L}(\mathcal{A})$ by swapping the left and the right children of all x_t-labeled transitions $q \xrightarrow{\boxed{x_t}} (q_0, q_1)$, i.e., update them to $q \xrightarrow{\boxed{x_t}} (q_1, q_0)$. We use $\mathrm{X}_t(\mathcal{A})$ to denote the TA constructed following this procedure.

Theorem 5.1. $\mathcal{L}(\mathrm{X}_t(\mathcal{A})) = \{\mathrm{X}_t(T) \mid T \in \mathcal{L}(\mathcal{A})\}$.

The update formulae of gates Z_t, S_t, and T_t are all in the form $a_1 \cdot B_{x_t} \cdot T + a_0 \cdot B_{\overline{x_t}} \cdot T$ for $a_1, a_0 \in \mathbb{C}$. Intuitively, the formulae scale the left and right subtrees of T with scalars a_0 and a_1, respectively. Their construction (Algorithm 1) can be done by (1) making one primed copy of $\mathcal{A}$ whose leaf labels are multiplied with a_1 (Line 3), (2) multiplying all leaf labels of $\mathcal{A}$ with a_0 (Line 4), and (3) updating all x_t-labeled transitions $q \xrightarrow{\boxed{x_t}} (q_0, q_1)$ to $q \xrightarrow{\boxed{x_t}} (q_0, q_1')$, i.e., for the right child, jump to the primed version (Line 4). In the algorithms, we define $Q' = \{q' \mid q \in Q\}$ for any set of state Q and $\Delta' = \{q' \xrightarrow{\boxed{x}} (q_l', q_r') \mid q \xrightarrow{\boxed{x}} (q_l, q_r) \in \Delta\}$ for any set of transitions Δ. The case of Y_t is similar, but we need both *constant scaling* (Lines 1-4) and *swapping* (Lines 7-9) (the left-hand side and right-hand side scalars being ω^2 and $-\omega^2$, respectively).

Theorem 5.2. $\mathcal{L}(\mathrm{U}(\mathcal{A})) = \{\mathrm{U}(T) \mid T \in \mathcal{L}(\mathcal{A})\}$, *for* $\mathrm{U} \in \{\mathrm{Y}_t, \mathrm{Z}_t, \mathrm{S}_t, \mathrm{T}_t\}$.

Algorithm 1: Algorithm for constructing $\mathrm{U}(\mathcal{A})$, for $\mathrm{U} \in \{X_t, Y_t, Z_t, S_t, T_t\}$

```
**Input:** A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$ and a gate U
**Output:** The TA $\mathrm{U}(\mathcal{A})$
1  **if** $\mathrm{U} \in \{Y_t, Z_t, S_t, T_t\}$ **then** // need constant scaling
2      Let $a_1$ and $a_0$ be the left and right scalar in $\mathrm{U}(T) = a_1 \cdot B_{x_t} \cdot T_1 + a_0 \cdot B_{\overline{x_t}} \cdot T_0$;
3      $\mathcal{A}_1 := \langle Q', \Sigma, \Delta_1, \mathcal{R}' \rangle$, where $\Delta_1 = \Delta'_i \cup \{q' \xrightarrow{\boxed{a_1 \cdot c}} () \mid q \xrightarrow{\boxed{c}} () \in \Delta_l\}$;
4      $\mathcal{A}^R := \langle Q \cup Q', \Sigma, \Delta^R \cup \Delta_1, \mathcal{R} \rangle$, where
           $\Delta^R = \{q \xrightarrow{\boxed{a_0 \cdot c}} () \mid q \xrightarrow{\boxed{c}} () \in \Delta_l\} \cup$
               $\{q \xrightarrow{\boxed{x_k}} (q_0, q_1) \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta_i \land k \neq t\} \cup$
               $\{q \xrightarrow{\boxed{x_k}} (q_0, q'_1) \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta_i \land k = t\}$
5  **else**
6      $\mathcal{A}^R := \mathcal{A}$; // when $\mathrm{U} = \mathrm{X}_t$
7  **if** $\mathrm{U} \in \{X_t, Y_t\}$ **then** // need swapping
8      Assume $\mathcal{A}^R = \langle Q^R, \Sigma, \Delta^R, \mathcal{R} \rangle$;
9      $\mathcal{A}^R := \langle Q^R, \Sigma, \Delta^R_1, \mathcal{R} \rangle$, where
           $\Delta^R_1 = \{q \xrightarrow{\boxed{x_k}} (q_0, q_1) \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta^R_i \land k \neq t\} \cup$
               $\{q \xrightarrow{\boxed{x_k}} (q_1, q_0) \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta^R_i \land k = t\} \cup \{t \mid t \in \Delta^R_l\}$
10 **return** $\mathcal{A}^R$;
```

The cases of multi-qubit gates CNOT^c_t, CZ^c_t, and $\mathrm{Toffoli}^{c,c'}_t$ can be handled when t is the lowest of the three qubits, i.e., $c < t \land c' < t$. We can assume w.l.o.g. that $c < c'$. Output of these gates can be constructed recursively following Algorithm 2. Let us look at the corresponding update formulae:

$$\mathrm{CNOT}^c_t(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})$$
$$\mathrm{CZ}^c_t(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T)$$
$$\mathrm{Toffoli}^{c,c'}_t(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_{c'}}} \cdot T + B_{x_{c'}} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}))$$

We first construct the TA of the inner term, the shaded area, which are TAs for X^t, Z^t, or $\mathrm{CNOT}^{c'}_t$. We call it the primed version here (cf. $\mathcal{A}'_1$ at Line 4). We then update all x_c-labeled transitions $q \xrightarrow{\boxed{x_c}} (q_0, q_1)$ to $q \xrightarrow{\boxed{x_c}} (q_0, q'_1)$, i.e., jump to the primed version in the right subtree.

Theorem 5.3. *$\mathcal{L}(\mathrm{U}(\mathcal{A})) = \{\mathrm{U}(T) \mid T \in \mathcal{L}(\mathcal{A})\}$, for $\mathrm{U} \in \{\mathrm{CNOT}^c_t, \mathrm{CZ}^c_t, \mathrm{Toffoli}^{c,c'}_t\}$.*

6 COMPOSITION-BASED ENCODING OF QUANTUM GATES

We introduce the composition-based approach in this section. The task is to develop TA operations that handle the update formulae in Table 1 compositionally. The idea is to lift the basic tree operations, such as projection T_{x_k}, restriction $B \cdot T$, and binary operation $\pm$ to operations over TAs and then compose them to have the desired gate semantics. The update formulae in Table 1 are always in the form of $\mathrm{term}_1 \pm \mathrm{term}_2$. For example, for the X_t gate, $\mathrm{term}_1 = B_{x_t} \cdot T_{\overline{x_t}}$ and $\mathrm{term}_2 = B_{\overline{x_t}} \cdot T_{x_t}$. Our idea is to first construct TAs $\mathcal{A}_{\mathrm{term}_1}$ and $\mathcal{A}_{\mathrm{term}_2}$, recognizing quantum states of term_1 and term_2, and then combine them using binary operation $\pm$ to produce a TA recognizing

Algorithm 2: Algorithm for constructing $\mathrm{U}(\mathcal{A})$, for $\mathrm{U} \in \{\mathrm{CNOT}_t^c, \mathrm{CZ}_t^c, \mathrm{Toffoli}_t^{c,c'}\}$

```
  Input: A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$ and a gate U
  Output: The TA $\mathrm{U}(\mathcal{A})$
1 if $\mathrm{U} = \mathrm{CNOT}_t^c$ then $\mathcal{A}_1 := \mathrm{X}_t(\mathcal{A})$;
2 if $\mathrm{U} = \mathrm{CZ}_t^c$ then $\mathcal{A}_1 := \mathrm{Z}_t(\mathcal{A})$;
3 if $\mathrm{U} = \mathrm{Toffoli}_t^{c,c'}$ then $\mathcal{A}_1 := \mathrm{CNOT}_t^{c'}(\mathcal{A})$;
4 Let $\mathcal{A}_1' = \langle Q_1', \Sigma, \Delta_1', \mathcal{R}' \rangle$ be obtained from $\mathcal{A}_1$ by priming all occurrences of states;
5 $\mathcal{A}^R := \langle Q \cup Q_1', \Sigma, \Delta^R \cup \Delta_1', \mathcal{R} \rangle$, where
      $\Delta^R = \{q \xrightarrow{\boxed{x_k}} (q_0, q_1) \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta_i \land k \neq c\} \cup$
                 $\{q \xrightarrow{\boxed{x_k}} (q_0, q_1') \mid q \xrightarrow{\boxed{x_k}} (q_0, q_1) \in \Delta_i \land k = c\} \cup \{t \mid t \in \Delta_l\}$
  return $\mathcal{A}^R$;
```

the quantum states of $\mathrm{term}_1 \pm \mathrm{term}_2$. The TAs $\mathcal{A}_{\mathrm{term}_1}$, $\mathcal{A}_{\mathrm{term}_2}$ would be constructed using TA versions of basic operations introduced later in this section.

For a TA accepting the trees $\{T_1, T_2\}$, a correct construction would produce a TA with the language $\{T_1' \pm T_1'', T_2' \pm T_2''\}$, for $T_i' = \mathrm{term}_1[T \mapsto T_i]$ and $T_i'' = \mathrm{term}_2[T \mapsto T_i]$, where $[T \mapsto T_i]$ is a substitution defined in the standard way. Obtaining this result is, however, not straightforward. If we just performed the $\pm$ operation pairwise between all elements of T_i' and T_i'', we would obtain the language $\{T_1' \pm T_1'', T_2' \pm T_2'', T_1' \pm T_2'', T_2' \pm T_1''\}$, which is wrong, since we are losing the information that T_1' and T_1'' are related (and so are T_2' and T_2'').

In the rest of the section, we will describe implementation of the necessary operations for the composition-based approach.

6.1 Tree Tag

We introduce the concept of *tree tags* to keep track of the origins of trees. For any tree T, its tag $\mathrm{Tag}(T)$ is the tree obtained from T by replacing all leaf symbols with a special symbol $\Box$. E.g., for the tree $T_1 = x_1(x_2(\mathbf{1}, \mathbf{0}), x_2(\mathbf{0}, \mathbf{0}))$, its tag is $\mathrm{Tag}(T_1) = x_1(x_2(\Box, \Box), x_2(\Box, \Box))$. Our construction needs to maintain the following invariants: (1) each tree in a TA has a unique tag, (2) all derived trees should have the same tag, and (3) binary operations over two sets of trees represented by TAs only combine trees with the same tag. When we say T' is *derived from* T, it means T' is obtained by applying basic tree operations on T. E.g., the tree $B_{\overline{x_1}} \cdot T_{x_1}$ is derived from T.

Example 6.1. Let $\mathcal{A}$ be a TA with root states $\mathcal{R} = \{q\}$ and transitions

$$q \xrightarrow{\boxed{x_1}} (q_l, q_r) \qquad q_l \xrightarrow{\boxed{x_2}} (q_1, q_0) \qquad q_0 \xrightarrow{\boxed{\mathbf{0}}} ()$$
$$q_l \xrightarrow{\boxed{x_2}} (q_0, q_1) \qquad q_1 \xrightarrow{\boxed{\mathbf{1}}} ()$$
$$q_r \xrightarrow{\boxed{x_2}} (q_0, q_0)$$

Observe that $\mathcal{L}(\mathcal{A}) = \{x_1(x_2(\mathbf{1}, \mathbf{0}), x_2(\mathbf{0}, \mathbf{0})), x_1(x_2(\mathbf{0}, \mathbf{1}), x_2(\mathbf{0}, \mathbf{0}))\}$. In Dirac notation, this is the set $\{|00\rangle, |01\rangle\}$. The tag of both trees is $x_1(x_2(\Box, \Box), x_2(\Box, \Box))$, which violates invariant (1) above. $\Box$

In general, invariant (1) does not hold, as we can see from Example 6.1. Our solution to this is introducing the *tagging* procedure (cf. Algorithm 3). The idea of tagging is simple: for each transition, we assign to its function symbol a unique number. After tagging a TA, every transition has a different symbol. Let $\mathrm{Untag}(T)$ be a function that removes the number j (added by the tagging procedure) from each symbol x_k^j in T's labels.

Algorithm 3: The tagging procedure Tag($\mathcal{A}$).

```
Input: A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$
Output: A tagged TA $\langle Q, \Sigma', \Delta', \mathcal{R} \rangle$
1 $\Delta_1 := \{q \xrightarrow{\boxed{c}} () \mid q \xrightarrow{\boxed{c}} () \in \Delta\}$;
2 $\Delta_2 := \{q \xrightarrow{\boxed{x_k^j}} (q_1, q_2) \mid \delta = (q \xrightarrow{\boxed{x_k}} (q_1, q_2)) \in \Delta, ord(\delta) = j\}$, where $ord\colon \Delta \to \mathbb{N}$ is an
    arbitrary injection (e.g., an ordering of the transitions);
3 $\Delta' := \Delta_1 \cup \Delta_2$;
4 $\Sigma'$ is the set of all symbols appearing in $\Delta'$;
5 return $\langle Q, \Sigma', \Delta', \mathcal{R} \rangle$;
```

Example 6.2. After tagging $\mathcal{A}$ from Example 6.1, we obtain the TA $\mathcal{A}_{\text{Tag}}$ with the root state q and the following transitions:

$$q \xrightarrow{\boxed{x_1^1}} (q_l, q_r) \qquad q_l \xrightarrow{\boxed{x_2^2}} (q_1, q_0) \qquad q_0 \xrightarrow{\boxed{\mathbf{0}}} ()$$
$$q_l \xrightarrow{\boxed{x_2^3}} (q_0, q_1) \qquad q_1 \xrightarrow{\boxed{\mathbf{1}}} ()$$
$$q_r \xrightarrow{\boxed{x_2^4}} (q_0, q_0)$$

Here $\mathcal{L}(\mathcal{A}_{\text{Tag}}) = \{T_1, T_2\}$, where $T_1 = x_1^1(x_2^2(\mathbf{1}, \mathbf{0}), x_2^4(\mathbf{0}, \mathbf{0}))$ and $T_2 = x_1^1(x_2^3(\mathbf{0}, \mathbf{1}), x_2^4(\mathbf{0}, \mathbf{0}))$. The two trees T_1 and T_2 have different tags now. □

Lemma 6.3. *All non-single-valued trees in a tagged TA have different tags.*

Definition 6.4 (Tag preservation). Given a tagged TA $\mathcal{A}_{\text{Tag}}$ and an operation U over binary trees, a TA construction procedure O transforming $\mathcal{A}_{\text{Tag}}$ to $O(\mathcal{A}_{\text{Tag}})$ is called *tag-preserving* if there is a bijection $S\colon \mathcal{L}(\mathcal{A}_{\text{Tag}}) \to \mathcal{L}(O(\mathcal{A}_{\text{Tag}}))$ such that $\text{Tag}(T) = \text{Tag}(S(T))$ for all $T \in \mathcal{L}(\mathcal{A}_{\text{Tag}})$. In such a case, we write $\mathcal{A}_{\text{Tag}} \simeq_{\text{Tag}} O(\mathcal{A}_{\text{Tag}})$. Further, if the above correspondence satisfies $U(\text{Untag}(T)) = \text{Untag}(S(T))$ for each T, we say that the TA construction procedure O is *tag-preserving over* U.

6.2 The Complete Picture of the Quantum Gate Application Procedure

Tagging a TA is the first step in applying a quantum gate. In the second step, for each term in the update formulae (cf. Table 1), we make a copy of the tagged TA and apply the operations that we are going to introduce (projection, restriction, and multiplication) to construct the corresponding TA. Notice that the operations are *tag-preserving*, i.e., they will keep the tag of all accepted trees. Then we use the binary operation to merge trees with the same tag and complete the update formula compositionally. In the end, we remove the TA's tag to finish the quantum gate application[4].

[4]This is a design choice. Another possibility is to keep the tag until finishing all gate operations. Untagging after finishing a gate has the advantage that it allows a more aggressive state space reduction.

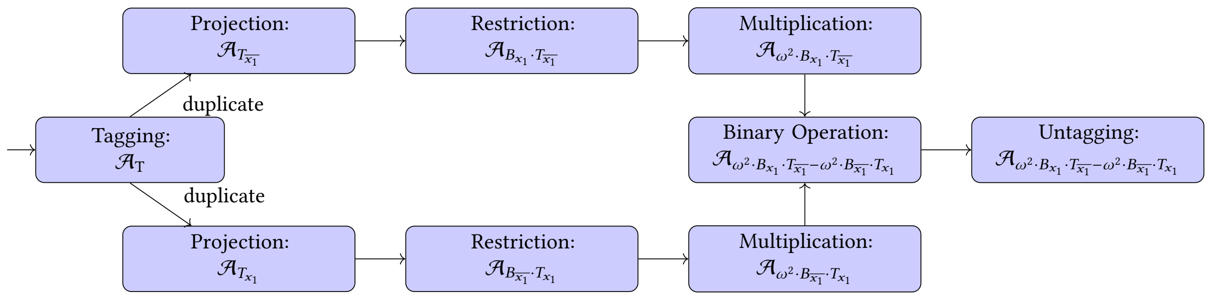

Projection: $\mathcal{A}_{T_{\overline{x_1}}}$
Restriction: $\mathcal{A}_{B_{x_1} \cdot T_{\overline{x_1}}}$
Multiplication: $\mathcal{A}_{\omega^2 \cdot B_{x_1} \cdot T_{\overline{x_1}}}$
duplicate
Tagging: $\mathcal{A}_{\text{T}}$
Binary Operation: $\mathcal{A}_{\omega^2 \cdot B_{x_1} \cdot T_{\overline{x_1}} - \omega^2 \cdot B_{\overline{x_1}} \cdot T_{x_1}}$
Untagging: $\mathcal{A}_{\omega^2 \cdot B_{x_1} \cdot T_{\overline{x_1}} - \omega^2 \cdot B_{\overline{x_1}} \cdot T_{x_1}}$
duplicate
Projection: $\mathcal{A}_{T_{x_1}}$
Restriction: $\mathcal{A}_{B_{\overline{x_1}} \cdot T_{x_1}}$
Multiplication: $\mathcal{A}_{\omega^2 \cdot B_{\overline{x_1}} \cdot T_{x_1}}$

Fig. 3. Constructions performed when applying the gate Y_1 to $\mathcal{A}_{\text{Tag}}$

Algorithm 4: Restriction operation on x_t, $\mathsf{Res}(\mathcal{A}, x_t, b)$

```
  Input: A tagged TA $\mathcal{A} = \{Q, \Sigma, \Delta, \mathcal{R}\}$,
  Output: A tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \{b \mathbin{?} B_{x_t} \cdot T \;:\; B_{\overline{x_t}} \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$
1 $\Delta'_i := \{q'_0 \xrightarrow{\boxed{x^i_j}} (q'_1, q'_2) \mid q_0 \xrightarrow{\boxed{x^i_j}} (q_1, q_2) \in \Delta\}$;
2 $\Delta'_l := \{q'_0 \xrightarrow{\boxed{\mathbf{0}}} () \mid q_0 \xrightarrow{\boxed{(a,b,c,d,k)}} () \in \Delta\}$;
3 $\Delta' := \Delta'_i \cup \Delta'_l$;
4 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{rm}} := \emptyset$;
5 foreach $q \xrightarrow{\boxed{x^i_t}} (q_l, q_r) \in \Delta$ do
6     if $b$ then $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{x^i_t}} (q'_l, q_r)\}$ else $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{x^i_t}} (q_l, q'_r)\}$;
7     $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{\boxed{x^i_t}} (q_l, q_r)\}$;
8 return $\{Q \cup Q', \Sigma \cup \{\mathbf{0}\}, ((\Delta \cup \Delta') \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\}$;
```

Example 6.5. From Table 1, we have

$$Y_1(T) = \omega^2 \cdot B_{x_1} \cdot T_{\overline{x_1}} - \omega^2 \cdot B_{\overline{x_1}} \cdot T_{x_1}.$$

For applying the gate Y_1 to a tagged TA $\mathcal{A}_{\mathrm{T}}$, we perform the constructions shown in Fig. 3. □

6.2.1 Restriction Operation: Constructing $\mathcal{A}_{B_{x_t} \cdot T}$ and $\mathcal{A}_{B_{\overline{x_t}} \cdot T}$ from $\mathcal{A}_T$. Observe that the tree $B_{x_t} \cdot T$ can be obtained by changing all leaf labels of the $\overline{x_t}$-subtrees in T to $(0, 0, 0, 0, 0)$. In Algorithm 4 we show the procedure for constructing the restriction operation based on this observation. Here $b \mathbin{?} s_1 \;:\; s_2$ is a shorthand for "if b is true then s_1 else s_2." Intuitively, when encountering a transition with variants of x_t as its label, in case $b = \mathbf{true}$, we reconnect its zero (left) child to the primed version (Line 6 of Algorithm 4), so the leaves of this subtree would be all zero. The case when $b = \mathbf{false}$ is symmetric. Note that the structure of the original and the primed versions are identical, so this modification will not change the tags of accepted trees.

Theorem 6.6. *Let $\mathcal{A}$ be a tagged TA. Then it holds that $\mathsf{Res}(\mathcal{A}, x_t, b) \simeq_{\mathrm{Tag}} \mathcal{A}$ and, moreover, $\mathcal{L}(\mathsf{Res}(\mathcal{A}, x_t, b)) = \{b \mathbin{?} B_{x_t} \cdot T \;:\; B_{\overline{x_t}} \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$.*

6.2.2 Multiplication Operation: Constructing $\mathcal{A}_{v \cdot T}$ from $\mathcal{A}_T$. Algorithm 5 gives the multiplication operation that works on both tagged and non-tagged version.

Algorithm 5: Multiplication operation, $\mathsf{Mult}(\mathcal{A}, v)$

```
  Input: A tagged TA $\mathcal{A} = \{Q, \Sigma, \Delta, \mathcal{R}\}$ and a constant value $v$ (either $\omega$ or $\frac{1}{\sqrt{2}}$)
  Output: A tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \{v \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$
1 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{rm}} := \emptyset$;
2 foreach $q \xrightarrow{\boxed{(a,b,c,d,k)}} () \in \Delta$ do
3     if $v = \omega$ then
4         $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{(-d,a,b,c,k)}} ()\}$;
5     else // $v = \frac{1}{\sqrt{2}}$
6         $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{(a,b,c,d,k+1)}} ()\}$;
7     $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{\boxed{(a,b,c,d,k)}} ()\}$;
8 return $\{Q, \Sigma, (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\}$;
```

Algorithm 6: Subtree copying procedure on x_t, s.copy$(\mathcal{A}, x_t, b)$.

```
Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$, variable $x_t$ to copy, and a Boolean value $b$ to indicate
       which branch to copy
Output: The tagged TA $\langle Q, \Sigma, \Delta', \mathcal{R} \rangle$
$\Delta_{\mathrm{rm}} := \Delta_{\mathrm{add}} := \emptyset$;
foreach $q \xrightarrow{\boxed{x_t^i}} (q_l, q_r) \in \Delta$ do
    if $b$ then $q_c := q_r$ else $q_c := q_l$;
    $\Delta_{\mathrm{add}} := \Delta_{\mathrm{add}} \cup \{q \xrightarrow{\boxed{x_t^i}} (q_c, q_c)\}$;
    $\Delta_{\mathrm{rm}} := \Delta_{\mathrm{rm}} \cup \{q \xrightarrow{\boxed{x_t^i}} (q_l, q_r)\}$;
return $\langle Q, \Sigma, (\Delta \setminus \Delta_{\mathrm{rm}}) \cup \Delta_{\mathrm{add}}, \mathcal{R} \rangle$;
```

Theorem 6.7. *Let $\mathcal{A}$ be a tagged TA. Then it holds that* $\mathsf{Mult}(\mathcal{A}, v) \simeq_{\mathrm{Tag}} \mathcal{A}$ *and, moreover,* $\mathcal{L}(\mathsf{Mult}(\mathcal{A}, v)) = \{v \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$.

6.2.3 Projection Operation: Constructing $\mathcal{A}_{T_{x_t}}$ and $\mathcal{A}_{T_{\overline{x_t}}}$ from $\mathcal{A}_T$. Recall that T_{x_t} is obtained from T by fixing the t-th input bit to be 1, i.e., $T_{x_t}(b_1 \dots b_t \dots b_n) = T(b_1 \dots 1 \dots b_n)$. Intuitively, the construction of $\mathcal{A}_{T_{x_t}}$ from $\mathcal{A}_T$ can be done by copying all right subtrees of x_t^i (i.e., corresponding to $x_t^i = 1$) to replace its left ($x_t^i = 0$) subtrees. A seemingly correct construction can be found in Algorithm 6. For short, we use s.copy$_{\mathrm{t}}(\mathcal{A})$ to denote s.copy$(\mathcal{A}, x_t, \mathbf{true})$ and s.copy$_{\bar{\mathrm{t}}}(\mathcal{A})$ to denote s.copy$(\mathcal{A}, x_t, \mathbf{false})$.

However, this construction has two issues (1) it would change the tag of accepting trees and (2) when there are more than one possible subtrees below q_r (or q_l), say, for example, T_1 and T_2, it might happen that the resulting TA accepts a tree such that one subtree below the symbol x_t^i is T_1 while another subtree is T_2, i.e., they are still not equal and hence not the result after copying.

Although the procedure is incorrect in general, it is correct when $t = n$, i.e., the layer just above the leaf. Notice that constant symbols are irrelevant to a tree's tag (all constant symbols will be replaced with $\square$ in a tag). So copying one subtree to the other will not affect the tag at the leaf transition. Moreover, recall that from TA's definition, all leaf transitions have unique starting states. So it will not encounter the issue (2) mentioned above.

Lemma 6.8. *Subtree copying* s.copy$_{\mathrm{t}}$ *is tag-preserving over the tree projection operation $T \longrightarrow T_{x_t}$ and* s.copy$_{\bar{\mathrm{t}}}$ *is tag-preserving over $T \longrightarrow T_{\overline{x_t}}$ when $t = n$.*

From the lemma above, we get the hint that the copy subtree procedure works only at the layer directly above leaf transitions, i.e., when applied to x_n. However, if we can reorder the variable without changing the set of quantum states encoded in a TA, then the projection procedure can be applied to any qubit. Below we will demonstrate a procedure for variable reordering (it is similar to a BDD variable reordering procedure [Felt et al. 1993]), but with an additional effort to preserve tree tags.

Example 6.9. Consider the following tree with the variable order $x_1 > x_2$

$$x_1(x_2(c_{00}, \mathbf{c_{01}}), x_2(\mathbf{c_{10}}, c_{11})),$$

here c_{ij} is the amplitude of $|ij\rangle$, which intuitively means x_1 takes value i and x_2 takes j. If we swap the variable order of the two variables, one can construct the tree below to capture the same quantum state

$$x_2(x_1(c_{00}, \mathbf{c_{10}}), x_1(\mathbf{c_{01}}, c_{11})).$$

Algorithm 7: Forward variable order swapping procedure on x_t, $\mathsf{f.swap}_t(\mathcal{A})$

```
  Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$
  Output: The tagged TA $\langle Q', \Sigma', \Delta', \mathcal{R} \rangle$
1 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{add}} := \emptyset, Q' := Q, \Sigma' := \Sigma$;
2 foreach $q \xrightarrow{\boxed{x_t^h}} (q_0, q_1), q_0 \xrightarrow{\boxed{x_l^i}} (q_{00}, q_{01}), q_1 \xrightarrow{\boxed{x_l^j}} (q_{10}, q_{11}) \in \Delta$ do
3     $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{x_l^{i,j}}} (q'_0, q'_1), q'_0 \xrightarrow{\boxed{x_t^h}} (q_{00}, \mathbf{q_{10}}), q'_1 \xrightarrow{\boxed{x_t^h}} (\mathbf{q_{01}}, q_{11})\}$;
4     $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{\boxed{x_t^h}} (q_0, q_1), q_0 \xrightarrow{\boxed{x_l^i}} (q_{00}, \mathbf{q_{01}}), q_1 \xrightarrow{\boxed{x_l^j}} (\mathbf{q_{10}}, q_{11})\}$;
5     $Q' := Q' \cup \{q'_0, q'_1\}$;
6     $\Sigma' := \Sigma' \cup \{x_l^{i,j}\}$;
7 return $\langle Q', \Sigma', (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R} \rangle$;
```

Notice the main difference of the two trees is that the two leaf labels c_{10} and c_{01} are swapped. This is because the second tree first picks the value of x_2 and then x_1, so the 01 node should be labeled c_{10}, which denotes x_1 takes value 1 and x_2 takes value 0. □

Inspired by the example, we can swap the order of two consecutive variables by modifying the transitions of a TA. One difficulty is that we want to keep trees' tags, so we introduce two procedures *forward variable order swapping* (Algorithm 7) and *backward variable order swapping* (Algorithm 8) to modify a variable's order while maintaining the trees' tag.

Algorithm 7 swaps the variable order of x_t and its succeeding symbol x_l, assuming the variable order is $\ldots > x_t > x_l > \ldots$. We assume that before running forward variable swapping, all symbols corresponding to qubits x_t and x_l are assigned unique numbers by the tagging procedure. After running the forward swapping procedure, we remember the unique numbers of both succeeding symbols x_l^i and x_l^j at the new upper layer's symbol $x_l^{i,j}$ (Line 3). So the trees' tag can be recovered in the backward variable order swapping procedure (Line 3 of Algorithm 8).

Then, the projection is computed as follows:

$$\mathsf{Prj}(\mathcal{A}, x_t, b) = \mathsf{b.swap}_t^{n-t}(\mathsf{s.copy}(\mathsf{f.swap}_t^{n-t}(\mathcal{A}), x_t, b)), \tag{13}$$

where a superscript i denotes repetition of the procedure i times. Each time when the forward swapping procedure is triggered, we move x_t^h one layer lower in all trees accepted by $\mathcal{A}$. We can move x_t^h to the layer above the leaf by repeatedly applying the forward swapping procedure, which fulfills the requirement for executing the subtree copying procedure. Then we use the backward swap procedure to return the variables to the original order. This procedure is potentially

Algorithm 8: Backward variable order swapping procedure on x_t, $\mathsf{b.swap}_t(\mathcal{A})$

```
  Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$
  Output: The tagged TA $\langle Q', \Sigma', \Delta', \mathcal{R} \rangle$
1 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{add}} := \emptyset, Q' := Q, \Sigma' := \Sigma$;
2 foreach $q \xrightarrow{\boxed{x_l^{i,j}}} (q'_0, q'_1), q'_0 \xrightarrow{\boxed{x_t^h}} (q_{00}, q_{10}), q'_1 \xrightarrow{\boxed{x_t^h}} (q_{01}, q_{11}) \in \Delta$ do
3     $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{\boxed{x_t^h}} (q''_0, q''_1), q''_0 \xrightarrow{\boxed{x_l^i}} (q_{00}, \mathbf{q_{01}}), q''_1 \xrightarrow{\boxed{x_l^j}} (\mathbf{q_{10}}, q_{11})\}$;
4     $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{\boxed{x_l^{i,j}}} (q'_0, q'_1), q'_0 \xrightarrow{\boxed{x_t^h}} (q_{00}, \mathbf{q_{10}}), q'_1 \xrightarrow{\boxed{x_t^h}} (\mathbf{q_{01}}, q_{11})\}$;
5     $Q' := Q' \cup \{q''_0, q''_1\}$;
6     $\Sigma' := \Sigma' \setminus \{x_l^{i,j}\}$;
7 return $\langle Q', \Sigma', (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R} \rangle$;
```

expensive, but TA minimization algorithms [Abdulla et al. 2008, 2007; Comon et al. 2008] can help to significantly reduce the cost.

Example 6.10. Here we demonstrate how the projection operations works with a concrete example. We assume that $\mathcal{A}$ is a tagged TA with the root state q and the following transitions:

$$\begin{array}{lll}
q \xrightarrow{\boxed{x_1^1}} (q_l, q_r) & q_l \xrightarrow{\boxed{x_2^2}} (q_1, q_0) & q_0 \xrightarrow{\boxed{\mathbf{0}}} () \\
 & q_l \xrightarrow{\boxed{x_2^3}} (q_0, q_1) & q_1 \xrightarrow{\boxed{\mathbf{1}}} () \\
 & q_r \xrightarrow{\boxed{x_2^4}} (q_0, q_0) &
\end{array}$$

Observe that $\mathcal{L}(\mathcal{A}) = \{T_1, T_2\}$, where

$$T_1 = x_1^1(x_2^2(\mathbf{1},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0})) \quad \text{and} \quad T_2 = x_1^1(x_2^3(\mathbf{0},\mathbf{1}), x_2^4(\mathbf{0},\mathbf{0})).$$

Then $\mathsf{f.swap}_k(\mathcal{A})$ produces a TA with a single root state q and the following transitions

$$\begin{array}{llll}
q \xrightarrow{\boxed{x_2^{2,4}}} (q_2, q_3) & q_2 \xrightarrow{\boxed{x_1^1}} (q_1, q_0) & q_4 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_0 \xrightarrow{\boxed{\mathbf{0}}} () \\
q \xrightarrow{\boxed{x_2^{3,4}}} (q_4, q_5) & q_3 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_5 \xrightarrow{\boxed{x_1^1}} (q_1, q_0) & q_1 \xrightarrow{\boxed{\mathbf{1}}} ()
\end{array}$$

The language $\mathcal{L}(\mathsf{f.swap}_k(\mathcal{A}))$ is $\{T_1', T_2'\}$, where

$$T_1' = x_2^{2,4}(x_1^1(\mathbf{1},\mathbf{0}), x_1^1(\mathbf{0},\mathbf{0})) \quad \text{and} \quad T_2' = x_2^{3,4}(x_1^1(\mathbf{0},\mathbf{0}), x_1^1(\mathbf{1},\mathbf{0})).$$

Note that T_1' and T_2' represent the same quantum states as T_1 and T_2 above. Then $\mathsf{s.copy}_1(\mathsf{f.swap}_1(\mathcal{A}))$ produces the following TA with the root state q:

$$\begin{array}{llll}
q \xrightarrow{\boxed{x_2^{2,4}}} (q_2, q_3) & q_2 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_4 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_0 \xrightarrow{\boxed{\mathbf{0}}} () \\
q \xrightarrow{\boxed{x_2^{3,4}}} (q_4, q_5) & q_3 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_5 \xrightarrow{\boxed{x_1^1}} (q_0, q_0) & q_1 \xrightarrow{\boxed{\mathbf{1}}} ()
\end{array}$$

Next we apply the backward swapping procedure to obtain $\mathcal{A}_{T_{x_1}}$, the final result of applying projection on $\mathcal{A}$. More concretely, $\mathcal{A}_{T_{x_1}} = \mathsf{b.swap}_1(\mathsf{s.copy}_1(\mathsf{f.swap}_1(\mathcal{A}))$ produces a TA with the root state q and the following transitions:

$$\begin{array}{llll}
q \xrightarrow{\boxed{x_1^1}} (q_2', q_3') & q_2' \xrightarrow{\boxed{x_2^2}} (q_0, q_0) & q_4' \xrightarrow{\boxed{x_2^3}} (q_0, q_0) & q_0 \xrightarrow{\boxed{\mathbf{0}}} () \\
q \xrightarrow{\boxed{x_1^1}} (q_4', q_5') & q_3' \xrightarrow{\boxed{x_2^4}} (q_0, q_0) & q_5' \xrightarrow{\boxed{x_2^4}} (q_0, q_0) & q_1 \xrightarrow{\boxed{\mathbf{1}}} ()
\end{array}$$

Observe that the language after projection is

$$\mathcal{L}(\mathcal{A}_{T_{x_1}}) = \{x_1(x_2^2(\mathbf{0},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0})), x_1(x_2^3(\mathbf{0},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0}))\},$$

which is the expected result. □

THEOREM 6.11. *Let $\mathcal{A}$ be a tagged TA. Then it holds that* $\mathsf{Prj}(\mathcal{A}, x_t, b) \simeq_{\mathrm{Tag}} \mathcal{A}$ *and, moreover,* $\mathcal{L}(\mathsf{Prj}(\mathcal{A}, x_t, b)) = \{b \mathbin{?} T_{x_t} : T_{\overline{x_t}} \mid T \in \mathcal{L}(\mathcal{A})\}$.

6.2.4 Binary Operation: $\mathcal{A}_{T_1 \pm T_2}$. Binary operation can be done by a modified product construction (cf. Algorithm 9). Notice that since we apply binary operations only over TAs derived from the same source TA, i.e., initially they have the same k at the leaf transitions, and the only possibility of changing the k part of a leaf symbol is the multiplication with $\frac{1}{\sqrt{2}}$, which is done only after all binary operations in Table 1, we can safely assume without loss of generality that $k_1 = k_2$.

THEOREM 6.12. *Let $\mathcal{A}_{T_1}$ and $\mathcal{A}_{T_2}$ be two tagged TAs. Then it holds that* $\mathcal{L}(\mathsf{Bin}(\mathcal{A}_1, \mathcal{A}_2, \pm)) = \{T_1 \pm T_2 \mid T_1 \in \mathcal{L}(\mathcal{A}_{T_1}) \wedge T_2 \in \mathcal{L}(\mathcal{A}_{T_2}) \wedge \mathsf{Tag}(T_1) = \mathsf{Tag}(T_2)\}$.

COROLLARY 6.13. *The composition-based encoding of quantum gate operations is correct.*

PROOF. Follows by Theorems 6.6, 6.7, 6.11 and 6.12. □

Algorithm 9: Binary operation, $\mathsf{Bin}(\mathcal{A}_1, \mathcal{A}_2, \pm)$

```
Input: Two tagged TAs $\mathcal{A}_1 = \langle Q_1, \Sigma, \Delta_1, \{q_1\}\rangle$ and $\mathcal{A}_2 = \langle Q_2, \Sigma, \Delta_2, \{q_2\}\rangle$.
Output: The tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \langle T_1 \pm T_2 \mid T_1 \in \mathcal{L}(\mathcal{A}_1) \wedge T_2 \in \mathcal{L}(\mathcal{A}_2)\rangle$
$\Delta'_i := \{(q^1, q^2) -\boxed{x^i_j}\!\!\to ((q^1_l, q^2_l), (q^1_r, q^2_r)) \mid q^1 -\boxed{x^i_j}\!\!\to (q^1_l, q^1_r) \in \Delta_1 \wedge q^2 -\boxed{x^i_j}\!\!\to (q^2_l, q^2_r) \in \Delta_2\}$;
$\Delta'_l := \{(q^1, q^2) -\boxed{(a_1 \pm a_2, b_1 \pm b_2, c_1 \pm c_2, d_1 \pm d_2, k_1)}\!\!\to () \mid q^1 -\boxed{(a_1, b_1, c_1, d_1, k_1)}\!\!\to () \in$
    $\Delta_1 \wedge q^2 -\boxed{(a_2, b_2, c_2, d_2, k_2)}\!\!\to () \in \Delta_2\}$;
return $\langle Q_1 \times Q_2, \Sigma', \Delta', \{(q_1, q_2)\}\rangle$;
```

7 EXPERIMENTAL EVALUATION

We implemented the proposed TA-based algorithm as a prototype tool named AutoQ in C++. We provide two settings: Hybrid, which uses the permutation-based approach (Section 5) to handle supported gates and switches to the composition-based approach for the other gates, and Composition, which handles all gates using the composition-based approach (Section 6). For checking language equivalence between the TA representing the set of reachable configurations and the TA for the post-condition, we use the Vata library [Lengál et al. 2012]. We use a lightweight simulation-based reduction [Bustan and Grumberg 2003] after finishing the Y, Z, S, T, CNOT, CZ, and Tofolli gate operations to keep the obtained TAs small.[5] All experiments were conducted on a server with an AMD EPYC 7742 64-core processor (1.5 GHz), 1,152 GiB of RAM (24 GiB for each process), and a 1 TB SSD running Ubuntu 20.04.4 LTS. Further details (pre- and post-conditions, circuits, etc.) can be found in [Chen et al. 2023b].

Data sets. We use the following set of benchmarks with quantum circuits:

- BV: Bernstein-Vazirani's algorithm with one hidden string of length n [Bernstein and Vazirani 1993],
- MCToffoli: circuits implementing multi-controlled Toffoli gates of size n using a variation of Nielsen and Chuang's decomposition [Nielsen and Chuang 2011] with standard Toffoli gates,
- Grover-Sing and Grover-All: implementation of Grover's search [Grover 1996] for a single oracle and for all possible oracles of length n (we encode the oracle's answer to be taken from the input; cf. [Chen et al. 2023b] for more details),
- FeynmanBench: 45 benchmarks from the tool suite Feynman [Amy 2018],
- RevLib: 80 benchmarks of reversible and quantum circuits [Wille et al. 2008], and
- Random: 20 randomly generated quantum circuits (10 circuits with 35 qubits and 105 gates and 10 circuits with 70 qubits and 210 gates).

We note that the benchmarks did not contain any unsupported gates.

Other tools. Since no existing work follows the same approach as we do, we compared AutoQ with representatives of the following approaches:

- *Quantum circuit simulators*: These compute the output of a quantum circuit for a given input quantum state. As a representative, we selected SliQSim [Tsai et al. 2021], a state-of-the-art quantum circuit simulator based on decision diagrams, which also works with a precise algebraic representation of complex numbers. We also tried the simulator from Qiskit [ANIS

[5]Our technique computes a non-maximum simulation by only checking whether states have the same successors. The results are in many cases the same as if the maximum simulation were computed, but the performance is much better. Further evaluation of this optimization of simulation is a future work.

Table 2. Verification of quantum algorithm. Here, n denotes the parameter value for the circuit, **#q** denotes the number of qubits, **#G** denotes the number of gates in the circuit. For AutoQ, the columns **before** and **after** have the format "states (transitions)" denoting the number of states and transitions in TA in the pre-condition and the output of our analysis respectively. The column **analysis** contains the time it took AutoQ to derive the TA for the output states and = denotes the time it took Vata to test equivalence. The timeout was 12 min. We use colours to distinguish the best result in each row and timeouts.

				AutoQ-Hybrid				AutoQ-Composition				SliQSim	Feynman	
	n	#q	#G	before	after	analysis	=	before	after	analysis	=	time	verdict	time
BV	95	96	241	193 (193)	193 (193)	6.0s	0.0s	193 (193)	193 (193)	7.1s	0.0s	0.0s	equal	0.5s
BV	96	97	243	195 (195)	195 (195)	5.9s	0.0s	195 (195)	195 (195)	7.1s	0.0s	0.0s	equal	0.5s
BV	97	98	246	197 (197)	197 (197)	6.3s	0.0s	197 (197)	197 (197)	7.4s	0.0s	0.0s	equal	0.6s
BV	98	99	248	199 (199)	199 (199)	6.5s	0.0s	199 (199)	199 (199)	7.7s	0.0s	0.0s	equal	0.6s
BV	99	100	251	201 (201)	201 (201)	6.7s	0.0s	201 (201)	201 (201)	7.8s	0.0s	0.0s	equal	0.6s
Grover-Sing	12	24	5,215	49 (49)	71 (71)	11s	0.0s	49 (49)	71 (71)	49s	0.0s	2.8s	timeout	
Grover-Sing	14	28	12,217	57 (57)	83 (83)	31s	0.0s	57 (57)	83 (83)	2m26s	0.0s	18s	timeout	
Grover-Sing	16	32	28,159	65 (65)	95 (95)	1m29s	0.0s	65 (65)	95 (95)	6m59s	0.0s	1m41s	timeout	
Grover-Sing	18	36	63,537	73 (73)	107 (107)	4m1s	0.0s	timeout				9m27s	timeout	
Grover-Sing	20	40	141,527	81 (81)	119 (119)	10m56s	0.0s	timeout				timeout	timeout	
MCToffoli	8	16	15	33 (42)	104 (149)	0.0s	0.0s	33 (42)	404 (915)	2.8s	0.0s	1.6s	equal	0.0s
MCToffoli	10	20	19	41 (52)	150 (216)	0.0s	0.0s	41 (52)	1,560 (3,607)	27s	0.0s	6.1s	equal	0.1s
MCToffoli	12	24	23	49 (62)	204 (295)	0.0s	0.0s	49 (62)	6,172 (14,363)	6m48s	0.1s	25s	equal	0.1s
MCToffoli	14	28	27	57 (72)	266 (386)	0.1s	0.0s	timeout				1m40s	equal	0.1s
MCToffoli	16	32	31	65 (82)	336 (489)	0.2s	0.0s	timeout				timeout	equal	0.2s
Grover-All	6	18	357	37 (43)	252 (315)	3.3s	0.0s	37 (43)	510 (573)	12s	0.0s	1.7s	timeout	
Grover-All	7	21	552	43 (50)	481 (608)	10s	0.0s	43 (50)	1,123 (1,250)	42s	0.0s	5.4s	timeout	
Grover-All	8	24	939	49 (57)	934 (1,189)	39s	0.1s	49 (57)	2,472 (2,727)	2m40s	0.0s	26s	timeout	
Grover-All	9	27	1,492	55 (64)	1,835 (2,346)	2m17s	0.4s	55 (64)	5,421 (5,932)	10m13s	0.1s	2m5s	timeout	
Grover-All	10	30	2,433	61 (71)	3,632 (4,655)	9m48s	2.1s	timeout				11m31s	timeout	

et al. 2021] (which does not provide a precise representation of numbers), but it was slower than SliQSim so we do not include it in the results.

- *Quantum circuit equivalence checkers*: We selected the following equivalence checkers: the verifier from the Feynman[6] tool suite [Amy 2018] (based on the path sum) and Qcec[7] [Burgholzer and Wille 2020] (combining decision diagrams, the ZX-calculus [Coecke and Duncan 2011], and random stimuli generation [Burgholzer et al. 2021]).

We evaluated AutoQ in two use cases, described in detail below.

7.1 Verification Against Pre- and Post-Conditions

In the first experiment, we compared how fast AutoQ computes the set of output quantum states and checks whether the set satisfies a given post-condition. We compared against the simulator SliQSim in the setting when we ran it over all states encoded in the pre-condition of the quantum algorithm and accumulated the times. We note that we did not include the time for comparing the result of SliQSim against a post-condition specification due to the following limitation of the tool: it can produce the state after executing the circuit in the vector form, but this step is not optimized and is quite time-consuming. Since the step of accumulating the obtained states could possibly be done in a more efficient way, avoiding transforming them first into the vector form, we do not include it in the runtime to not give SliQSim an unfair disadvantage. The timeout was 12 min.

We also include the time taken by Feynman to check the equivalence of the circuits with themselves. Although checking equivalence of quantum circuits is a harder problem than what we are solving (so the results cannot be used for direct comparison with AutoQ), we include these results in order to give an idea about hardness of the circuits for path-sum-based approaches.

[6] Git commit 56e5b771
[7] Version 2.0.0

Table 3. Results for bug finding. The notation is the same as in Table 2. In addition, the column **bug?** indicates if the tool caught the injected bug: T denotes that the bug was found, F denotes that the tool gave an incorrect result, and — means unknown result (includes the tool reporting *unknown*, crash, or not enough resources). AutoQ finds all bugs within the time limit, and we provide the number of iterations needed to catch the bug (column **iter**). The timeout was 30 min.

				AutoQ		Feynman		Qcec	
	circuit	#q	#G	time	iter	time	bug?	time	bug?
FeynmanBench	csum_mux_9	30	141	0.8s	1	6.5s	—	44.0s	F
	gf2^10_mult	30	348	2.0s	1	0.6s	—	42.7s	F
	gf2^16_mult	48	876	11s	1	4.8s	—	58.5s	T
	gf2^32_mult	96	3,323	2m4s	1	48.1s	—	1m58s	T
	ham15-high	20	1,799	8.0s	1	3m51s	—	30.2s	T
	mod_adder_1024	28	1,436	10s	1	9.2s	—	31.9s	T
	hwb10	16	31,765	1m42s	1	timeout		30.2s	T
	hwb11	15	87,790	4m23s	1	timeout		35.9s	T
	hwb12	20	171,483	13m43s	1	timeout		1m3s	T
	hwb8	12	6,447	15s	1	timeout		23.4s	T
	qcla_adder_10	36	182	2.8s	1	1.3s	—	46.6s	F
	qcla_mod_7	26	295	2.6s	1	1m24s	—	38.4s	F
Random	35a	35	106	3.2s	1	0.2s	—	45.7s	F
	35b	35	106	1.4s	1	0.2s	T	47.8s	F
	35c	35	106	1.3s	1	0.2s	T	47.5s	T
	35d	35	106	1.3s	1	0.2s	T	48.2s	T
	35e	35	106	1.3s	1	0.1s	—	50.6s	T
	35f	35	106	2.4s	1	0.3s	T	49.7s	F
	35g	35	106	4.0s	3	0.2s	—	55.3s	T
	35h	35	106	1.0s	1	0.2s	—	0.6s	—
	35i	35	106	1.3s	1	0.2s	T	54.8s	T
	35j	35	106	1.8s	1	0.2s	—	51.4s	F
	70a	70	211	16s	1	1.1s	—	1m18s	T
	70b	70	211	14s	1	0.8s	T	1m11s	T
	70c	70	211	12s	1	0.9s	—	1m24s	T
	70d	70	211	29m29s	36	1.2s	T	1m26s	T
	70e	70	211	17s	1	1.0s	—	1m30s	T
	70f	70	211	33s	1	0.9s	T	1m26s	F
	70g	70	211	14m42s	44	1.2s	—	1m35s	T
	70h	70	211	13s	1	1.2s	—	1m36s	T
	70i	70	211	23s	1	1.2s	—	1m36s	T
	70j	70	211	2m5s	1	1.4s	—	1m34s	T
RevLib	add16_174	49	65	2.6s	1	timeout		1m8s	T
	add32_183	97	129	17s	1	timeout		2m4s	T
	add64_184	193	257	1m55s	1	timeout		0.6s	—
	avg8_325	320	1,758	21m18s	1	timeout		0.5s	—
	bw_291	87	308	10s	1	11.7s	T	1m55s	T
	cycle10_293	39	79	0.5s	1	0.4s	T	1m7s	T
	e64-bdd_295	195	388	36s	1	timeout		0.5s	—
	ex5p_296	206	648	1m52s	1	1m29s	T	0.4s	—
	ham15_298	45	154	0.6s	1	0.6s	T	1m14s	T
	mod5adder_306	32	97	0.5s	1	0.7s	T	1m1s	T
	rd84_313	34	105	0.5s	1	1.1s	T	1m2s	T
	urf1_149	9	11,555	30s	1	timeout		35.8s	T
	urf2_152	8	5,031	11s	1	21m33s	T	32.5s	T
	urf3_155	10	26,469	1m19s	1	timeout		33.0s	T
	urf4_187	11	32,005	1m57s	1	timeout		31.4s	T
	urf5_158	9	10,277	27s	1	timeout		26.6s	T
	urf6_160	15	10,741	1m6s	1	timeout		36.2s	T
	hwb6_301	46	160	2.0s	1	2.7s	T	1m7s	T
	hwb7_302	73	282	8.3s	1	10.9s	T	1m38s	T
	hwb8_303	112	450	27s	1	37.9s	T	2m22s	T
	hwb9_304	170	700	1m33s	1	2m20s	T	0.6s	—

We ran this experiment on the benchmarks where the semantics was known to us so that we could construct TAs with pre- and post-conditions. These were the following: BV, MCToffoli, Grover-Sing, and Grover-All. We give the results in Table 2. Both BV and Grover-Sing work with only one input state, which should be most favourable for simulators. Surprisingly, for the case of Grover-Sing, AutoQ outperforms SliQSim on large cases (out of curiosity, we tried to run SliQSim on Grover-Sing (n=20) without a timeout; the running time was 51m43s). We attribute the good performance of AutoQ to the compactness of the TA representation of Grover's state space. On the other hand, both MCToffoli and Grover-All consider 2^n input states and we can observe the exponential factor emerging; hence AutoQ outperforms SliQSim in large cases. All tools perform pretty well on BV, even cases with 100 qubits can be easily handled. We can also see that Hybrid is consistently faster than Composition.

7.2 Finding Bugs

In the following experiment, we compared AutoQ with the equivalence checkers Feynman and Qcec and evaluated the ability of the tools to determine that two quantum circuits are non-equivalent (this is to simulate the use case of verifying the output of an optimizer). We took circuits from the benchmarks FeynmanBench, Random, and RevLib, and for each circuit, we created a copy and injected an artificial bug (one additional randomly selected gate at a random location). Then we ran the tools and let them check circuit equivalence; for AutoQ, we let it compute two TAs representing sets of output states for both circuits for the given set of input states and then checked their language equivalence with Vata.

Our strategy for finding bugs with AutoQ (we used the Hybrid setting) was the following: We started with a TA representing a single basis state, i.e., a TA with no top-down nondeterminism, and gradually added more non-deterministic transitions (in each iteration one randomly chosen transition) into the TA, making it represent a larger set of states, running the analysis for each of the TAs, until we found the bug. This proved to be a successful strategy, since running the analysis with an input TA representing, e.g., all possible basis states, might be too challenging (generally speaking, the larger is the TA representing the set of states, the slower is the analysis).

We present the results in Table 3. We exclude trivial cases (all tools can finish within 5 s) and difficult cases that no tool can handle within the timeout period (30 min). We can see that many of the cases were so tricky that equivalence checkers failed to conclude anything, while AutoQ succeeded in finding the bug with just the first few TAs. For two instances from Random (70d and 70g), we found the bug after trying 36 TAs after 29m29s and 44 TAs after 14m42s, respectively. For a few cases (e.g., csum_mux_9), Qcec did not find the bug and reported that the circuits were equivalent (F)[8], while AutoQ reported it (T). For these cases, we fed the witness produced by AutoQ to SliQSim and confirmed the two circuits are different.

The results show that our approach to hunting for bugs in quantum circuits is beneficial, particularly for larger circuits where equivalence checkers do not scale. For such cases, AutoQ can still find bugs using a weaker specification. For instance, AutoQ was able to find bugs in some large-scale instances from RevLib with hundreds of qubits, e.g., add64_184 and avg_8_325, while both Feynman and Qcec fail.

We note that the area of quantum circuit equivalence checking is rapidly advancing. When preparing the final version, we became aware of SliQEC [Chen et al. 2022; Wei et al. 2022], a recent tool that outperforms the other equivalence checkers that we tried on this benchmark.

8 RELATED WORK

Circuit equivalence checkers are often very efficient but less flexible in specifying the desired property (only equivalence). Our approach can switch to a lightweight specification when verification fails due to insufficient resources and still find bugs in the design. Often equivalence checking is done by a reduction to normal form using a set of rewriting rules. *Path-sum* is a recent approach proposed in [Amy 2018], whose rewrite rules can solve the equivalence problem of Clifford group circuits in polynomial time. The *ZX-calculus* [Coecke and Duncan 2011] is a graphical language that is particularly useful in circuit optimization and proving equivalence. The works of [Hietala et al. 2019] ensures correctness of the rewrite rules with a theorem prover. Quartz [Xu et al. 2022b] is a circuit optimization framework consisting of an equivalence checker based on some precomputed equivalence sets. We pick Feynman [Amy 2018], a state-of-the-art equivalence checker based on path-sum, and Qcec [Burgholzer and Wille 2020], based on decision diagrams and ZX-calculus, as the baseline tools for comparison. *Quantum circuit simulators*, e.g. SliQSim [Tsai et al. 2021], can be used as equivalence checkers for a finite number of inputs by trying all basis states.

Quantum abstract interpretation [Perdrix 2008; Yu and Palsberg 2021] is particularly efficient in processing large-scale circuits, but it over-approximates state space and cannot conclude anything when verification fails. For instance, the work in [Yu and Palsberg 2021] can only distinguish quantum states with zero and non-zero probability (and cannot derive exact boundary probabilities). In contrast, our approach precisely represents reachable states and can reveal bugs. One can consider our approach to be an instantiation of classical abstract interpretation [Cousot and Cousot 1977] that is precise, and our approach to non-equivalence testing as comparing output abstract contexts of two programs. *Quantum model checking* supports a rich specification language (flavors of temporal logic [Feng et al. 2013; Mateus et al. 2009; Xu et al. 2022a]). It can be seen as an extension of probabilistic model checking [Feng et al. 2017, 2015, 2013; Xu et al. 2022a; Ying 2021; Ying and Feng 2021; Ying et al. 2014] and is more suitable for verifying high-level protocols due to the

[8]This bug has been confirmed by the Qcec team and fixed later, cf. [QCE 2022].

limited scalability [Anticoli et al. 2016]. Techniques based on *quantum simulation* [Green et al. 2013; Niemann et al. 2016; Pednault et al. 2017; Samoladas 2008; Tsai et al. 2021; Viamontes et al. 2009; Wecker and Svore 2014; Zulehner et al. 2019; Zulehner and Wille 2019] allow only one input quantum state and thus have limited analyzing power.

Quantum Hoare logic [Feng and Ying 2021; Liu et al. 2019; Unruh 2019; Ying 2012; Zhou et al. 2019]) allows verification against complex correctness properties and rich program constructs such as branches and loops, but requires significant manual work. On the other hand, *quantum incorrectness logic* [Yan et al. 2022] is a dual of quantum Hoare logic that allows showing the existence of a bug, but cannot prove its absence. The Qbricks [Chareton et al. 2021] approach alleviates the difficulty of proof search by combining state-of-the-art theorem provers with decision procedures, but, according to their experiments, still requires a significant amount of human intervention. For instance, their experiments show that it requires 125 times intervention during verification of Grover's search w.r.t. an arbitrary number of qubits.

9 CONCLUDING REMARKS

We have introduced a new paradigm for quantum circuit analysis that is exciting from both practical and theoretical lenses. We demonstrated one of its potential applications—circuit non-equivalence checking, but we believe there could be much more. In our own experience of using the method to prepare the benchmarks, its role is similar to a static assertion checker (like *software model checkers* for classical programs [Chen et al. 2016; Heizmann et al. 2018]); it helped us greatly to find several problems while composing the circuits. The connection to automata-based verification is also quite exciting. A series of approaches from the classical world should also be helpful in the quantum case. For instance, the idea of regular tree model checking could be leveraged to verify parameterized quantum circuits (w.r.t. an arbitrary number of qubits) [Abdulla et al. 2002; Bouajjani et al. 2012]. For this, one would need to deal with TAs with loops, where tagging cannot be done anymore to impose relations among trees (one would need to use an unbounded number of tags)—new ideas are needed. Automata-learning can be used for automatic loop invariant inference [Chen et al. 2017a]. Symbolic automata [D'Antoni and Veanes 2017] and register automata [Chen et al. 2017b] would allow using variables to describe amplitude (instead of a fixed alphabet as we use now). We believe there are many other techniques from the automata world that could be used to extend our framework and be applied in the area of analysing quantum circuits.

ACKNOWLEDGMENTS

We thank the reviewers for their in-depth remarks that helped us improve the quality of the paper and the artifact committee members for their helpful suggestions about the artifact. This material is based on a work supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme; the Czech Science Foundation project GA23-07565S; the FIT BUT internal project FIT-S-23-8151; and the NSTC QC project under Grant no. NSTC 111-2119-M-001-004-.

DATA AVAILABILITY STATEMENT

An environment with the tools and data used for the experimental evaluation in the current study is available at [Chen et al. 2023a].

REFERENCES

2022. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/
2022. The Qcec repository: Issue #200 (ZX-Checker produces invalid result). https://github.com/cda-tum/qcec/issues/200
Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati, and Tomás Vojnar. 2008. Computing Simulations over Tree Automata. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*

2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (LNCS, Vol. 4963), C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 93–108. https://doi.org/10.1007/978-3-540-78800-3_8

Parosh Aziz Abdulla, Johanna Högberg, and Lisa Kaati. 2007. Bisimulation Minimization of Tree Automata. *Int. J. Found. Comput. Sci.* 18, 4 (2007), 699–713. https://doi.org/10.1142/S0129054107004929

Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. 2002. Regular Tree Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (LNCS, Vol. 2404)*. Springer, 555–568. https://doi.org/10.1007/3-540-45657-0_47

Dorit Aharonov. 2003. A Simple Proof that Toffoli and Hadamard are Quantum Universal. https://doi.org/10.48550/arxiv.quant-ph/0301040

Thorsten Altenkirch and Jonathan Grattage. 2005. A Functional Quantum Programming Language. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 249–258. https://doi.org/10.1109/LICS.2005.1

Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). 1–21. https://doi.org/10.4204/EPTCS.287.1

Matthew Amy. 2019. *Formal Methods in Quantum Circuit Design*. Ph. D. Dissertation. University of Waterloo.

MD SAJID ANIS, Abby-Mitchell, Héctor Abraham, et al. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. 2016. Towards Quantum Programs Verification: From Quipper Circuits to QPMC. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings (LNCS, Vol. 9720)*, Simon J. Devitt and Ivan Lanese (Eds.). Springer, 213–219. https://doi.org/10.1007/978-3-319-40578-0_16

Frank Arute et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (Oct. 2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5 Number: 7779 Publisher: Nature Publishing Group.

Ethan Bernstein and Umesh V. Vazirani. 1993. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 11–20. https://doi.org/10.1145/167088.167097

Jacob D. Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202. https://doi.org/10.1038/nature23474

Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. 2012. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer* 14, 2 (2012), 167–191. https://doi.org/10.1007/s10009-011-0205-y

Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. 2000. Regular Model Checking. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (LNCS, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 403–418. https://doi.org/10.1007/10722167_31

P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani P. Roychowdhury, and Farrokh Vatan. 2000. A new universal and fault-tolerant quantum basis. *Inf. Process. Lett.* 75, 3 (2000), 101–107. https://doi.org/10.1016/S0020-0190(00)00084-3

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

Lukas Burgholzer, Richard Kueng, and Robert Wille. 2021. Random Stimuli Generation for the Verification of Quantum Circuits. In *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 18-21, 2021*. ACM, 767–772. https://doi.org/10.1145/3394885.3431590

Lukas Burgholzer and Robert Wille. 2020. Advanced equivalence checking for quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (2020), 1810–1824. https://doi.org/10.1109/TCAD.2020.3032630

Doron Bustan and Orna Grumberg. 2003. Simulation-based minimazation. *ACM Trans. Comput. Log.* 4, 2 (2003), 181–206. https://doi.org/10.1145/635499.635502

Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. 2019. Quantum Chemistry in the Age of Quantum Computing. *Chemical Reviews* 119, 19 (2019), 10856–10915. https://doi.org/10.1021/acs.chemrev.8b00803 arXiv:https://doi.org/10.1021/acs.chemrev.8b00803 PMID: 31469277.

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-Building Quantum Programs. In *ESOP (LNCS, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

Tian-Fu Chen, Jie-Hong R. Jiang, and Min-Hsiu Hsieh. 2022. Partial Equivalence Checking of Quantum Circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 594–604. https://doi.org/10.1109/QCE53715.2022.00082

Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023a. *An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits.* https://doi.org/10.5281/zenodo.7811406

Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023b. An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits (Technical Report). (2023). arXiv:2301.07747 [cs.LO]

Yu-Fang Chen, Chih-Duo Hong, Anthony W Lin, and Philipp Rümmer. 2017a. Learning to prove safety over parameterised concurrent systems. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 76–83. https://doi.org/10.23919/FMCAD.2017.8102244

Yu-Fang Chen, Chiao Hsieh, Ondřej Lengál, Tsung-Ju Lii, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. 2016. PAC learning-based verification and model synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. 714–724. https://doi.org/10.1145/2884781.2884860

Yu-Fang Chen, Ondřej Lengál, Tony Tan, and Zhilin Wu. 2017b. Register automata with linear arithmetic. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005111

Carlo Ciliberto, Mark Herbster, Alessandro Davide Ialongo, Massimiliano Pontil, Andrea Rocchetto, Simone Severini, and Leonard Wossnig. 2018. Quantum Machine Learning: A Classical Perspective. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 474, 2209 (January 2018). https://doi.org/10.1098/rspa.2017.0551

Bob Coecke and Ross Duncan. 2011. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics* 13, 4 (apr 2011), 043016. https://doi.org/10.1088/1367-2630/13/4/043016

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree automata techniques and applications.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. 2015. Fast: A Transducer-Based Language for Tree Manipulation. *ACM Trans. Program. Lang. Syst.* 38, 1 (2015), 1:1–1:32. https://doi.org/10.1145/2791292

Christopher M. Dawson and Michael A. Nielsen. 2006. The Solovay-Kitaev algorithm. *Quantum Inf. Comput.* 6, 1 (2006), 81–95. https://doi.org/10.26421/QIC6.1-6

Loris D'Antoni and Margus Veanes. 2017. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*. Springer, 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

Mark Ettinger, Peter Høyer, and Emanuel Knill. 2004. The quantum query complexity of the hidden subgroup problem is polynomial. *Inf. Process. Lett.* 91, 1 (2004), 43–48. https://doi.org/10.1016/j.ipl.2004.01.024

Andrew Fagan and Ross Duncan. 2019. Optimising Clifford Circuits with Quantomatic. *Electronic Proceedings in Theoretical Computer Science* 287 (jan 2019), 85–105. https://doi.org/10.4204/eptcs.287.5

Eric Felt, Gary York, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. 1993. Dynamic variable reordering for BDD minimization. In *Proceedings of the European Design Automation Conference 1993, EURO-DAC '93 with EURO-VHDL'93, Hamburg, Germany, September 20-24, 1993.* IEEE Computer Society, 130–135. https://doi.org/10.1109/EURDAC.1993.410627

Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Shenggang Ying. 2017. Model checking omega-regular properties for quantum Markov chains. In *28th International Conference on Concurrency Theory (CONCUR 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.CONCUR.2017.35

Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. 2015. QPMC: A Model Checker for Quantum Programs and Protocols. In *International Symposium on Formal Methods*, Nikolaj Bjørner and Frank de Boer (Eds.). Springer International Publishing, 265–272. https://doi.org/10.1007/978-3-319-19249-9_17

Yuan Feng and Mingsheng Ying. 2021. Quantum Hoare logic with classical variables. *ACM Transactions on Quantum Computing* 2, 4 (2021), 1–43. https://doi.org/10.1145/3456877

Yuan Feng, Nengkun Yu, and Mingsheng Ying. 2013. Model checking quantum Markov chains. *J. Comput. Syst. Sci.* 79, 7 (2013), 1181–1198. https://doi.org/10.1016/j.jcss.2013.04.002

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 333–342. https://doi.org/10.1145/2491956.2462177

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 212–219. https://doi.org/10.1145/237814.237866

Wakaki Hattori and Shigeru Yamashita. 2018. Quantum Circuit Optimization by Changing the Gate Order for 2D Nearest Neighbor Architectures. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings (LNCS, Vol. 11106)*, Jarkko Kari and Irek Ulidowski (Eds.). Springer, 228–243. https://doi.org/10.1007/978-3-319-99498-7_16

Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. 2018. Ultimate Automizer and the search for perfect interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019. Verified optimization in a quantum intermediate representation. *arXiv preprint arXiv:1904.06319* (2019).

Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. 2020. Optimization of quantum circuit mapping using gate transformation and commutation. *Integr.* 70 (2020), 43–50. https://doi.org/10.1016/j.vlsi.2019.10.004

Dominik Janzing, Pawel Wocjan, and Thomas Beth. 2005. "Non-Identity-Check" Is QMA-complete. *International Journal of Quantum Information* 03, 03 (2005), 463–473. https://doi.org/10.1142/S0219749905001067

Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. 2012. VATA: A library for efficient manipulation of non-deterministic tree automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 79–94. https://doi.org/10.1007/978-3-642-28756-5_7

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal verification of quantum algorithms using quantum Hoare logic. In *International conference on computer aided verification*. Springer, 187–207. https://doi.org/10.1007/978-3-030-25543-5_12

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

Paul Massey, John A. Clark, and Susan Stepney. 2005. Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, Hans-Georg Beyer and Una-May O'Reilly (Eds.). ACM, 1657–1663. https://doi.org/10.1145/1068009.1068288

Paulo Mateus, Jaime Ramos, Amílcar Sernadas, and Cristina Sernadas. 2009. *Temporal Logics for Reasoning about Quantum Systems*. Cambridge University Press, 389–413. https://doi.org/10.1017/CBO9781139193313.011

Nikolaj Moll, Panagiotis Barkoutsos, Lev S Bishop, Jerry M Chow, Andrew Cross, Daniel J Egger, Stefan Filipp, Andreas Fuhrer, Jay M Gambetta, Marc Ganzhorn, Abhinav Kandala, Antonio Mezzacapo, Peter Müller, Walter Riess, Gian Salis, John Smolin, Ivano Tavernelli, and Kristan Temme. 2018. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Science and Technology* 3, 3 (jun 2018), 030503. https://doi.org/10.1088/2058-9565/aab822

Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4 (2018). Issue 23. https://doi.org/10.1038/s41534-018-0072-4

Daniel Neider and Nils Jansen. 2013. Regular Model Checking Using Solver Technologies and Automata Learning. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (LNCS, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 16–31. https://doi.org/10.1007/978-3-642-38088-4_2

Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, USA.

Philipp Niemann, Robert Wille, D. Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. 2016. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35, 1 (2016), 86–99. https://doi.org/10.1109/TCAD.2015.2459034

Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W. Draeger, Eric T. Holland, and Robert Wisnieff. 2017. Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral. *CoRR* abs/1710.05867 (2017). http://arxiv.org/abs/1710.05867

Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence checking paradigms in quantum circuit design: a case study. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 517–522. https://doi.org/10.1145/3489517.3530480

Simon Perdrix. 2008. Quantum entanglement analysis based on abstract interpretation. In *International Static Analysis Symposium*. Springer, 270–282. https://doi.org/10.1007/978-3-540-69166-2_18

Vasilis Samoladas. 2008. Improved BDD Algorithms for the Simulation of Quantum Circuits. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings (LNCS, Vol. 5193)*, Dan Halperin and Kurt Mehlhorn (Eds.). Springer, 720–731. https://doi.org/10.1007/978-3-540-87744-8_60

Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

Mathias Soeken, Robert Wille, Gerhard W. Dueck, and Rolf Drechsler. 2010. Window optimization of reversible and quantum circuits. In *13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2010, Vienna, Austria, April 14-16, 2010*. IEEE Computer Society, 341–345. https://doi.org/10.1109/DDECS.2010.5491754

Lee Spector. 2006. Automatic Quantum Computer Programming: A Genetic Programming Approach. (2006).

Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. 2021. Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 439–444. https://doi.org/10.1109/DAC18074.2021.9586191

Dominique Unruh. 2019. Quantum Hoare logic with ghost variables. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785779

George F. Viamontes, Igor L. Markov, and John P. Hayes. 2007. Checking equivalence of quantum circuits and states. In *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, Georges G. E. Gielen (Ed.). IEEE Computer Society, 69–74. https://doi.org/10.1109/ICCAD.2007.4397246

George F. Viamontes, Igor L. Markov, and John P. Hayes. 2009. *Quantum Circuit Simulation*. Springer. https://doi.org/10.1007/978-90-481-3065-8

Dave Wecker and Krysta M. Svore. 2014. LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing. *CoRR* abs/1402.4467 (2014). arXiv:1402.4467 http://arxiv.org/abs/1402.4467

Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R. Jiang. 2022. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 523–528. https://doi.org/10.1145/3489517.3530481

R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*. 220–225. https://doi.org/10.1109/ISMVL.2008.43 RevLib is available at http://www.revlib.org.

Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1234–1240. https://doi.org/10.23919/DATE.2019.8715261

Ming Xu, Jianling Fu, Jingyi Mei, and Yuxin Deng. 2022a. Model checking QCTL plus on quantum Markov chains. *Theor. Comput. Sci.* 913 (2022), 43–72. https://doi.org/10.1016/j.tcs.2022.01.044

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. 2022b. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 625–640. https://doi.org/10.1145/3519939.3523433

Shigeru Yamashita and Igor L. Markov. 2010. Fast equivalence-checking for quantum circuits. *Quantum Inf. Comput.* 10, 9&10 (2010), 721–734. https://doi.org/10.26421/QIC10.9-10-1

Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. On incorrectness logic for Quantum programs. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–28. https://doi.org/10.1145/3527316

Mingsheng Ying. 2012. Floyd-Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49. https://doi.org/10.1145/2049706.2049708

Mingsheng Ying. 2021. Model Checking for Verification of Quantum Circuits. In *International Symposium on Formal Methods*. Springer, 23–39. https://doi.org/10.1007/978-3-030-90870-6_2

Mingsheng Ying and Yuan Feng. 2021. *Model Checking Quantum Systems: Principles and Algorithms*. Cambridge University Press.

Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. 2014. Model-checking linear-time properties of quantum systems. *ACM Transactions on Computational Logic (TOCL)* 15, 3 (2014), 1–31. https://doi.org/10.1145/2629680

Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H Ibarra. 2008. Symbolic string verification: An automata-based approach. In *International SPIN Workshop on Model Checking of Software*. Springer, 306–324. https://doi.org/10.1007/978-3-540-85114-1_21

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924. https://doi.org/10.1142/S0129054111009112

Nengkun Yu and Jens Palsberg. 2021. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 542–558. https://doi.org/10.1145/3453483.3454061

Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1149–1162. https://doi.org/10.1145/3314221.3314584

Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942057

Alwin Zulehner and Robert Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427

Received 2022-11-10; accepted 2023-03-31

Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization

Tian-Fu Chen
d11k42001@ntu.edu.tw
Grad. School of Advanced Technology
National Taiwan University
Taipei, Taiwan

Yu-Fang Chen
gulu0724@gmail.com
Institute of Information Science
Academia Sinica
Taipei, Taiwan

Jie-Hong Roland Jiang
jhjiang@ntu.edu.tw
Grad. Inst. of Electronics Engineering
National Taiwan University
Taipei, Taiwan

Sára Jobranová
xjobra01@stud.fit.vutbr.cz
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Ondřej Lengál
lengal@fit.vutbr.cz
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

ABSTRACT

Quantum circuit simulation is the basic tool for reasoning over quantum programs. Despite the tremendous advance in the simulator technology in the recent years, the performance of simulators is still unsatisfactory on non-trivial circuits, which slows down the development of new quantum systems. In this work, we develop a loop summarizing simulator based on multi-terminal binary decision diagrams (MTBDDs) with efficiently customized quantum gate operations. The simulator is capable of automatic loop summarization using symbolic execution, which saves repetitive computation for circuits with iterative structures. Experimental results show the simulator outperforms state-of-the-art simulators on some standard circuits, such as Grover's algorithm, by several orders of magnitude.

ACM Reference Format:
Tian-Fu Chen, Yu-Fang Chen, Jie-Hong Roland Jiang, Sára Jobranová, and Ondřej Lengál. 2024. Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization . In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676711

1 INTRODUCTION

The development of quantum computers started in 1980s with the promise to solve problems challenging for classical computers. Later, quantum algorithms more efficient than their best classical counterparts for certain problems started appearing, such as Shor's algorithm for integer factoring [27] or Grover's algorithm for search in an unstructured database [19]. With multiple major players investing into quantum and the consistent improvement of the hardware, it seems that quantum computers will occupy a prominent role in the future. The development of quantum algorithms is an extremely challenging task so adequate computer-aided support is needed for debugging and reasoning over quantum programs.

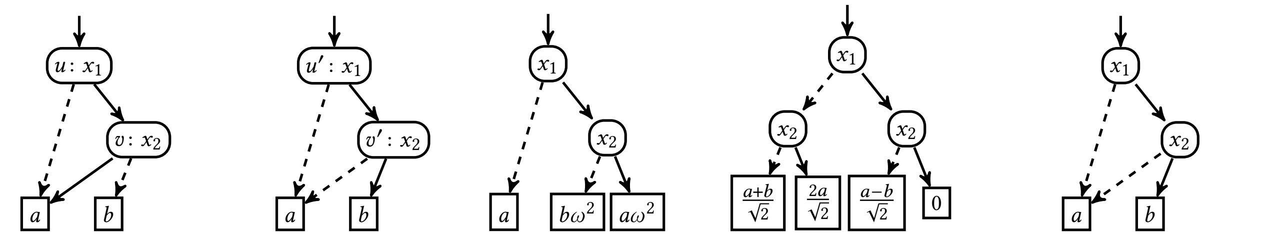
Debugging quantum programs is primarily done through *simulation*, which is considerably more challenging in the quantum world as compared to the classical world. This is because, in the quantum world, we need to keep track of a potentially exponentially sized *quantum state* that assigns *every* classical state a complex *amplitude* instead of keeping track of a *single* evolving classical program state.

Simulators of quantum programs have advanced tremendously in recent years, moving from the basic vector- and matrix-based representation [26] into representations based on decision diagrams [25, 28, 30, 32, 33, 36], graphical languages [14], or model counting [24]. Despite this advance, simulating quantum circuits of a moderate size is still considered infeasible. Therefore, faster simulators are needed to provide quantum developers with basic means to observe behaviour of quantum programs.

In this paper, we focus on accelerating the simulation of quantum circuits that contain repetition of some sub-structure. Some notable examples of such circuits include *Grover's search* [19], *period finding* [23], and *quantum counting* [8]. Current standards for describing quantum circuits, such as the OpenQASM 3.0 format [15], allow describing such repeated sub-structures compactly using loops or hierarchical gate definitions.

Our method for accelerating simulation involves computing a *symbolic summary* of a sequence of quantum gates that occur repeatedly, such as a loop body or the definition of a hierarchical gate. This summary is computed with respect to a particular quantum state and can be reused to execute the sequence of quantum gates from any state that shares the same high-level structure, i.e., computational bases with the same amplitudes in the first state will also have the same amplitudes in the second state, though these amplitude values may differ from those in the first state. We derive these summaries using *symbolic execution*, which is similar to standard quantum simulation but instead computes symbolic terms that remember the arithmetic operations to be performed, rather than computing the results of arithmetic operations over numbers.

Moreover, similarly to [30], we represent quantum states algebraically for exact simulation without numerical precision loss, which is crucial in tasks such as equivalence checking [34]. Unlike [30], which works only for concrete value simulation, ours



(a) MTBDD M_q for q **(b) Applying X_2 to q** **(c) Applying S_1 to q** **(d) Applying H_1 to q** **(e) Applying CNOT_2^1 to q**

Figure 1: Examples of applying quantum gates on MTBDD-based representation of the state $q = a\,|00\rangle + a\,|01\rangle + b\,|10\rangle + a\,|11\rangle$.

allows symbolic simulation thanks to the use of *multi-terminal binary decision diagrams* (MTBDDs) [4, 9, 16]. We customize MTBDD procedures for efficient quantum gate execution instead of using only standard MTBDD functions *Apply* and *Restrict* as usual.

Our experimental evaluation shows that our proposed approach can significantly speed up simulation for some well-established quantum circuits. This allows us to tackle circuits of sizes that were previously considered infeasible.

2 PRELIMINARIES

$\mathbb{B} = \{0, 1\}$ denotes the Booleans. We fix a set $\mathbb{X} = \{x_1, \ldots, x_n\}$ of Boolean variables with an order $x_1 < x_2 < \cdots < x_n$; we use $\vec{x}$ to denote $(x_1, \ldots, x_n)$. Given an arbitrary set $S \neq \emptyset$, a *pseudo-Boolean* function is a function $f\colon \mathbb{B}^n \to S$; f is a *Boolean* function if $S = \mathbb{B}$. ω denotes the complex number $e^{\frac{i\pi}{4}}$, i.e., the unit vector that makes an angle of 45° with the positive real axis in the complex plane.

2.1 Decision diagrams

Given an arbitrary nonempty set S with finitely representable elements, a *multi-terminal binary decision diagram* (MTBDD) [4, 9, 16] is a graph $G = (N, T, \mathit{low}, \mathit{high}, \mathit{root}, \mathit{var})$ where N is the set of *internal nodes*, $T \subseteq S$ is the set of *leaf nodes* ($T \cap N = \emptyset$, $T \neq \emptyset$), $\mathit{low}, \mathit{high}\colon N \to (N \cup T)$ are the *low*- and *high*-successor edges, $\mathit{root} \in N \cup T$ is the *root* node, and $\mathit{var}\colon N \to \mathbb{X}$ is the node-variable mapping, with the following three restrictions:

(i) (connectivity) every node from $N \cup T$ is reachable from *root* over some sequence of *low* and *high* edges,
(ii) (order) for every $u, v \in N$, if $\mathit{low}(u) = v$ or $\mathit{high}(u) = v$, then $\mathit{var}(u) < \mathit{var}(v)$, and
(iii) (reducedness) there is no node $u \in N$ s.t. $\mathit{low}(u) = \mathit{high}(u)$.

Each node $v \in N \cup T$ represents a pseudo-Boolean function $[\![v]\!]$ defined inductively as follows: (1) if $v \in T$, then $[\![v]\!](\vec{x}) = v$, and (2) if $v \in N$ and $\mathit{var}(v) = x_i$, then

$$[\![v]\!](\vec{x}) = \begin{cases} [\![\mathit{low}(v)]\!](\vec{x}) & \text{if } x_i = 0 \text{ and} \\ [\![\mathit{high}(v)]\!](\vec{x}) & \text{if } x_i = 1. \end{cases}$$

Moreover, we impose the following additional restriction on G:

(iv) (canonicity) there are no two nodes $u \neq v$ such that $[\![u]\!] = [\![v]\!]$.

G then represents the function $[\![G]\!]$ defined as $[\![\mathit{root}]\!]$. We abuse notation and confuse a function with the MTBDD representing it and use a node r to denote the MTBDD rooted in r and vice versa.

We will use the following standard MTBDD operations. The `apply`(f_1, f_2, op_2) operation is used to combine two MTBDDs f_1 and f_2 through a binary operation $op_2\colon S \times S \to S$ performed on the corresponding leaf notes, obtaining the MTBDD representing the pseudo-Boolean function $\{\vec{x} \mapsto op_2(f_1(\vec{x}), f_2(\vec{x})) \mid \vec{x} \in \mathbb{B}^n\}$. The `monadic_apply`$(f, op)$ operation updates the leaves of the MTBDD f with a unary operation $op_1\colon S \to S$, obtaining the MTBDD representing the pseudo-Boolean function $\{\vec{x} \mapsto op_1(f(\vec{x})) \mid \vec{x} \in \mathbb{B}^n\}$. We often use lambda expression for defining $op_{1/2}$. Additionally, MTBDDs provide the `spawn`(l, h, x) function that works as follows: (i) if $l = h$, then the result is l,otherwise (ii) the result is the unique node n such that $\mathit{low}(n) = l$, $\mathit{high}(n) = h$, and $\mathit{var}(n) = x$.

2.2 Quantum Computing Fundamentals

Quantum computers are programmed through *quantum gates*, which update the global *quantum state*. A *quantum circuit* is a sequence of gates, combined with programming constructs like *loops* or *hierarchical gate definitions* that allow a more concise presentation [15].

Quantum states: In a traditional computer system with n bits, a state is represented by n Booleans. In the quantum world, such states are called *computational basis states*. E.g., in a system with three bits labeled x_1, x_2, and x_3, the computational basis state $|011\rangle$ indicates that the value of x_1 is 0 and the values of x_2 and x_3 are 1.

In a quantum system, an n-qubit *quantum state* is a probabilistic distribution over n-bit basis states, denoted either as a column vector $(a_0, \ldots, a_{2^n-1})^T$ (given here as a transposed row vector) or as a formal sum $\sum_{j \in \{0,1\}^n} a_j \cdot |j\rangle$, where $a_0, a_1, \ldots, a_{2^n-1} \in \mathbb{C}$ are *complex amplitudes* satisfying the property that $\sum_{j \in \{0,1\}^n} |a_j|^2 = 1$. Intuitively, $|a_j|^2$ is the probability that when we measure the quantum state in the computational basis, we obtain the classical state $|j\rangle$; these probabilities must sum up to 1 for all basis states. We can view a quantum state as a function mapping each basis state in $\mathbb{B}^n$ to a complex amplitude and represent them using MTBDDs; cf. Figure 1a for an MTBDD M_q representing the state $q = a\,|00\rangle + a\,|01\rangle + b\,|10\rangle + a\,|11\rangle$ (for some $a, b \in \mathbb{C}$ s.t. $a \neq b$ and $3|a|^2 + |b|^2 = 1$).

Quantum gates: Two main types of quantum gates are being used: *single-qubit gates* and *controlled gates*. We support all commonly used gates except the arbitrary rotation single-qubit gate due to the use a precise complex number representation (cf. Sec. 5).

Single-qubit gates. In general, a single-qubit gate is presented as a *unitary complex matrix*. We directly support the following gates:

$$\mathrm{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \mathrm{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \mathrm{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

$$\mathrm{S} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad \mathrm{T} = \begin{pmatrix} 1 & 0 \\ 0 & \omega \end{pmatrix}, \quad \mathrm{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$\mathrm{R_X}\left(\frac{\pi}{2}\right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}, \quad \mathrm{R_Y}\left(\frac{\pi}{2}\right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}.$$

Algorithm 1: Execution of a single-qubit gate U_t

```
   Input: MTBDD $M_q = (N, T, low, high, root, var)$,
          target qubit $x_t$, single qubit gate U
   Output: MTBDD representing $\mathrm{U}_t(M_q)$
1  return recurse($root$);
2  Function recurse(node)
3      $l \leftarrow low(\text{node}); h \leftarrow high(\text{node}); x_i \leftarrow var(\text{node})$;
4      if $i < t$ then
5          $l_{new} \leftarrow$ recurse($l$); $h_{new} \leftarrow$ recurse($h$);
6          return spawn($l_{new}, h_{new}, x_i$);
7      else // $i \geq t$ or a leaf
8          if $i = t$ then $l' \leftarrow l; h' \leftarrow h$ ;
9          else $l' \leftarrow h' \leftarrow$ node;
10         if $\mathrm{U} = \mathrm{X}$ then return spawn($h', l', x_t$) ;
11         if $\mathrm{U} \in \{\mathrm{T}, \mathrm{S}, \mathrm{Z}\}$ then
12             if $U = T$ then $c \leftarrow \omega$;
13             if $U = S$ then $c \leftarrow \omega^2$;
14             if $U = Z$ then $c \leftarrow -1$;
15             $h_{new} \leftarrow$monadic_apply($h', \lambda x(c \cdot x)$);
16             return spawn($l', h_{new}, x_t$);
17         if $\mathrm{U} = \mathrm{Y}$ then
18             $l_{new} \leftarrow$monadic_apply($h', \lambda x(-\omega^2 \cdot x)$);
19             $h_{new} \leftarrow$monadic_apply($l', \lambda x(\omega^2 \cdot x)$);
20             return spawn($l_{new}, h_{new}, x_t$);
21         if $\mathrm{U} = \mathrm{H}$ then
22             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y))$);
23             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y))$);
24             return spawn($l_{new}, h_{new}, x_t$);
25         if $\mathrm{U} = \mathrm{R_X}(\frac{\pi}{2})$ then
26             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - \omega^2 \cdot y))$);
27             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (y - \omega^2 \cdot x))$);
28             return spawn($l_{new}, h_{new}, x_t$);
29         if $\mathrm{U} = \mathrm{R_Y}(\frac{\pi}{2})$ then
30             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y))$);
31             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y))$);
32             return spawn($l_{new}, h_{new}, x_t$);
```

For a single-qubit gate U, we often use a subscript to denote the qubit that it is applied to, e.g., U_i means we apply U to qubit x_i.

The X gate is the quantum "*negation*" gate. Applying gate X to a single-qubit state $\binom{l}{h}$ produces the state $\mathrm{X} \cdot \binom{l}{h} = \binom{h}{l}$. In the case of an MTBDD-based representation of $\binom{l}{h}$, which would have a root node with the *low*-successor $l \in T$ and *high*-successor $h \in T$, this would effectively mean swapping the *low* and *high* successors of the root. For the general case, applying X_i to a quantum state's MTBDD swaps the high and low-successor edges of all nodes at level i. See Figure 1b for an example of applying X_2 to the MTBDD M_q introduced above (the edges leaving v' got swapped).

Algorithm 2: Execution of a controlled gate CU_t^c

```
   Input: MTBDD $M = (N, T, low, high, root, var)$,
          control qubit $x_c$, target qubit $x_t$, single qubit gate U
   Output: MTBDD representing $\mathrm{CU}_t^c(M_q)$
1  $M_l \leftarrow$ recurse($root$, L);
2  $M_h \leftarrow$ recurse($\mathrm{U}_t(M_q)$, H);
3  return apply($M_l, M_h, \lambda x, y(x + y)$);
4  Function recurse(node, dir)
5      $l \leftarrow low(\text{node}); h \leftarrow high(\text{node}); x_i \leftarrow var(\text{node})$;
6      if $i < c$ then
7          $l_{new} \leftarrow$ recurse($l$, $dir$); $h_{new} \leftarrow$ recurse($h$, $dir$);
8          return spawn($l_{new}, h_{new}, x_i$);
9      else // $i \geq c$ or a leaf
10         if $i = c$ then $l' \leftarrow l; h' \leftarrow h$ ;
11         else $l' \leftarrow h' \leftarrow$ node;
12         if dir = L then return spawn($l'$, 0, $x_c$) ;
13         else return spawn(0, $h'$, $x_c$) ;
```

Behaviours of Z, S, and T gates are similar to each other. In particular, applying the gates to $\binom{l}{h}$ produces the states $\mathrm{Z} \cdot \binom{l}{h} = \binom{l}{-h}$, $\mathrm{S} \cdot \binom{l}{h} = \binom{l}{i \cdot h}$, and $\mathrm{T} \cdot \binom{l}{h} = \binom{l}{\omega \cdot h}$, which multiply the $|1\rangle$-position with -1, i, and ω, respectively. Similarly, applying Z, S, and T to a quantum state's MTBDD multiplies all leaves in the *high*-subtrees of all nodes at level i with -1, i, and ω, respectively (cf. Figure 1c for an example of applying S_1 to M_q).

The last group of single-qubit gates we mention includes H (the *Hadamard* gate), $\mathrm{R_X}(\frac{\pi}{2})$, and $\mathrm{R_Y}(\frac{\pi}{2})$. These gates are more challening for implementation, since they fuse the amplitudes of the two basis states to form a new state. Taking H as an example, it updates the state $\binom{l}{h}$ to the state $H \cdot \binom{l}{h} = \frac{1}{\sqrt{2}} \cdot \binom{l+h}{l-h}$. See Figure 1d for the result of applying H_1 to M_q. We refer the readers to Sec. 3 for the corresponding MTBDD constructions.

Controlled gates. A controlled gate CU uses another quantum gate U as its parameter. We often use CU_t^c to denote applying the controlled-gate with control qubit x_c and target qubit x_t. The effect of the controlled-U gate is that the gate U_t is applied only when the control qubit x_c has the value 1. For example, the controlled-X gate CNOT_2^1 has the control qubit x_1 and would apply X_2 when x_1 is valued 1. See Figure 1e for an example of applying CNOT_2^1 to M_q.

3 ALGORITHM FOR QUANTUM GATES

Single-qubit gates. In Algorithm 1, we present our procedure for applying *single-qubit gates* to an MTBDD $M_q = (N, T, low, high, root, var)$ at the target qubit x_t. The procedure performs the operations on M_q directly, as opposed to the standard approach (used, e.g., in SliQSim [30]), which uses only the standard (MT)BDD interface (in particular, functions *Apply* and *Restrict*).

The algorithm is as a modification of a standard `monadic_apply`. In particular, it performs a depth-first search (Line 5) until it reaches an x_t node, then it performs the semantic of the gate on the successors. The semantic differs for the particular gate, and was already briefly discussed in Sec. 2.2. We, however, need to be careful about *"don't care"* edges, i.e., edges that skip some variable in the MTBDD (such as the *low* edge from u in Figure 1a). In such a situation, we need to stop the recursion and perform the gate operation by materializing the missing node (with the same *low* and *high*, cf. Line 9. E.g., when applying X_2 to the state q in Figure 1a, we have $l' = h' = a$ when handling the *low*-successor of u. Calling $\mathtt{spawn}(a, a, x_2)$ will just return the a leaf. On the other hand, $high(u')$ will be set to $\mathtt{spawn}(high(v), low(v), x_2) = \mathtt{spawn}(a, b, x_2) = v'$.

To apply T, S, and Z gates, we use `monadic_apply` to multiply the leaf nodes of *high*-successors of the nodes labelled by x_t with ω, ω^2, and -1, respectively. When applying S_1, one step would be computing $\mathtt{monadic\_apply}(v, \lambda x(\omega^2 \cdot x))$ and connecting the result to *high* of the new root via the `spawn` function (Figure 1c). Meanwhile, the Y gate does for each node at level i the following: (1) it multiplies the *high* with $-\omega^2$ and sets it as the new *low*, and (2) it multiplies the *low* with ω^2 and sets it as the new *high*.

For each node at level i, applying the H, $R_X(\frac{\pi}{2})$, or $R_Y(\frac{\pi}{2})$ gates merges the *high* and *low*-successors using the `apply` function, creating new *high* and *low*-successors according to the gate's behaviour. In the case of the H gate, the new *low*-successor is $\mathtt{apply}(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y)))$ and the new *high*-successor is $\mathtt{apply}(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y)))$. When applying H_1 to the state q, we have $h' = v$ and $l' = a$. Fusing the two via $\mathtt{apply}(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x+y)))$ gives us the *low*-successor of the root in Figure 1d and via $\mathtt{apply}(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y)))$ gives us the *high*-successor of the root.

Controlled gates. Our procedure for applying *controlled*-U *gates* to M_q at the control qubit x_c for some quantum gate U is presented in Algorithm 2. The procedure involves three steps. First, in M_l, we will store a copy of M_q modified such that every base with $x_c = 1$ has amplitude 0 (Line 1). Second, we compute an MTBDD $U_t(M)$ using some of Algorithms 1 and 2 (depending on U, which can again be a controlled gate) and modify it such that every base with $x_c = 0$ has amplitude 0 (Line 2). Finally, both MTBDDs are summed up using the `apply` function (Line 3), which will, effectively, combine the two MTBDDs together (one operand of the + is always 0). Note that the *Toffoli* gate can be obtained by using the CNOT gate for U. A specialized more efficient version of the algorithm for phase gates (e.g., Z, S, T) can be used (omitted here due to space constraints).

Memoization. In order to avoid redundant computation, calls to the `recurse` functions in Algorithms 1 and 2 should be memoized.

Concrete execution and symbolic execution. Our gate operations work for both concrete and symbolic amplitude values. When leaf values are concrete, e.g., when $x = \frac{1}{2}$ and $y = \frac{1}{4}$, the function $\lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y)))$ will compute the value $\frac{1}{\sqrt{2}} \cdot (\frac{1}{2} + \frac{1}{4}) = \frac{3}{4\sqrt{2}}$. When leaf values are symbolic, e.g, , when $x = x_0$ and $y = y_0$, the same function will compute the symbolic term $\frac{1}{\sqrt{2}} \cdot (x_0 + y_0))$.

4 LOOP SUMMARIZATION

Our main contribution is an optimization that targets algorithms with loops[1], such as various *amplitude amplification* algorithms [7], with the most famous one being Grover's unstructured search [19]. The optimization is particularly effective in the case that the number of distinct amplitudes is small (which is the case for amplitude amplification algorithms, where there are typically only a limited number of different amplitudes at the beginning of a loop body, e.g., high amplitude, low amplitude, and zero).

[1]W.l.o.g., in the basic version of the optimization presented here, we assume the loop bodies are unitaries, i.e., do not contain measurements, and that they are not nested.

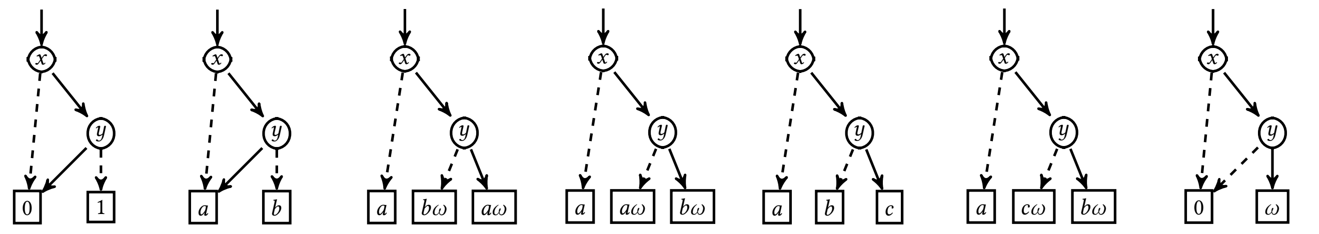
Algorithm 3: Loop summarization

```
   Input: An MTBDD $M_q$, a loop body $C$
   Output: An MTBDD $M_\alpha$ over $\mathbb{S}$ and a mapping $\tau\colon \mathbb{S} \to \mathbb{T}_\mathbb{S}$
1  $\alpha \leftarrow \emptyset$ (type $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$);            // init abstraction
2  $M_\alpha^{refined} \leftarrow$ monadic_apply($M_q$, abstract[$\alpha$]);
3  repeat
4      $M_\alpha \leftarrow M_\alpha^{refined}$;
5      $M'_\alpha \leftarrow C_S(M_\alpha)$;
6      $\tau \leftarrow \emptyset$ (type $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}$);               // update
7      $\sigma \leftarrow \emptyset$ (type $\sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}$);        // refinement subst
8      $M_\alpha^{refined} \leftarrow$ apply($M_\alpha$, $M'_\alpha$, refine[$\tau, \sigma, \alpha$]);
9  until $M_\alpha = M_\alpha^{refined}$;
10 return $(M_\alpha, \tau)$;
11 Function abstract(val)
       Data: $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
12     if $\alpha$(val) $= \bot$ then
13         let $s_{new} \in \mathbb{S} \setminus \mathrm{rng}(\alpha)$ be a fresh symbolic var.;
14         $\alpha \leftarrow \alpha \cup \{\mathrm{val} \mapsto s_{new}\}$;
15     return $\alpha$(val);
16 Function refine(lhs, rhs)
       Data: $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}, \sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}, \alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
17     if $\tau$(lhs) $= \bot$ then
18         $\tau \leftarrow \tau \cup \{\mathrm{lhs} \mapsto \mathrm{rhs}\}$;
19     else if $\not\models \tau(\mathrm{lhs}) = \mathrm{rhs}$ then
20         if $\sigma$(lhs) $= \bot$ then
21             let $s_{new} \in \mathbb{S} \setminus \mathrm{rng}(\alpha)$ be a fresh symbolic var.;
22             $\sigma \leftarrow \sigma \cup \{\mathrm{lhs} \mapsto s_{new}\}$;
23         return $\sigma$(lhs);
24     return lhs;
```

Intuitively, the optimization works as follows. Consider a circuit with the following loop (in the OpenQASM 3.0 format [15]):

```
for int i in [1:K] { C; }
```

where C is the unitary for the loop body composed of standard gates and K is a constant. When a simulation of the circuit arrives to the loop with a quantum state q represented by MTBDD M_q, it will first create an MTBDD M_α with leaves containing symbolic variables (from a set $\mathbb{S}$, an infinite set of symbolic names). Then, it will run circuit C of the loop body with M_α as its input, with operations being done symbolically, i.e., instead of numbers, the leaves of the resulting MTBDD M'_α contain terms over $\mathbb{S}$; we denote the set of terms over $\mathbb{S}$ as $\mathbb{T}_\mathbb{S}$. M'_α contains information about how each of the computational bases needs to be updated. The information in M'_α is, however, fine-tuned for M_q, which can make the representation quite compact. This fine-tuning is done in the initial step called *abstraction*, when symbolic variables are being introduced—we start by introducing one symbolic variable for every distinct leaf value in M_q. The assumption is that computational bases with the same value will behave similarly. This does not need to hold, so after M'_α is



(a) Concrete state M_q **(b) Initial abstraction M_{α_1} of M_q** **(c) After applying T_x to M_{α_1}** **(d) End of first iteration M'_{α_1}** **(e) Refined abstraction M_{α_2} of M_q** **(f) End of second iteration M'_{α_2}** **(g) Result of M_q after M'_{α_2}**

Figure 2: An example run of Algorithm 3 on the circuit in Figure 3.

computed, we check it by observing whether bases mapping to the same symbolic variable in M_α also map to the same update in M'_α. If not, we introduce more symbolic variables (for the differing bases) and run the algorithm again, until the condition holds.

The formal algorithm is given in Algorithm 3. In the algorithm, we use the following formal notation: $f[p_1, \ldots, p_k]$ denotes the closure of function f with parameters $p_1, \ldots, p_k$ assigned to the variables in the **Data** declaration of f (passed by reference). Given a (partial) function f of the type $f : X \rightharpoonup Y$, we use $\mathrm{rng}(f)$ to denote the *range* of f, i.e., the set $\{y \in Y \mid \exists x \in X : f(x) = y\}$. Moreover, given an $x \in X$, if there is no $(x, y) \in f$, we write $f(x) = \bot$.

Example 1. We first demonstrate a run of the algorithm on the example circuit in Figure 3. The circuit starts in state q with $x = 1$ and $y = 0$. Then, it performs K executions of the loop body C. In each execution of the loop body, first, the T gate is applied to x, performing the multiplication of its $|1\rangle$ amplitude by ω and then CNOT of y controlled by x is performed. Therefore, the resulting state after K executions is $K\omega\,|11\rangle$ if K is odd and $K\omega\,|10\rangle$ if K is even. The run of Algorithm 3 on the circuit is shown in Figure 2.

M_q is in Figure 2a. In Figure 2b, we can see the initial abstraction M_{α_1} of M_q after Line 2; in this case, $\alpha_1 = \{0 \mapsto a, 1 \mapsto b\}$ for symbolic variables a and b. Then, we run (Line 5) the loop body with M_{α_1}, obtaining first the tree in Figure 2c (after T^x) and then the tree M'_{α_1} in Figure 2d (after CNOT^x_y). Then, when we call $\texttt{apply}(M_{\alpha_1}, M'_{\alpha_1}, \texttt{refine}[\tau_1, \sigma_1, \alpha_1])$ at Line 8, we realize that the inital abstraction α_1 was too coarse (going from left to right, we will construct $\tau_1 = \{a \mapsto a, b \mapsto a\omega\}$ for bases $|00\rangle$, $|01\rangle$, and $|10\rangle$; then, when processing $|11\rangle$, which would give us $a \mapsto b\omega$, which is in conflict with $a \mapsto a$, we will introduce a new symbolic variable c for the base $|11\rangle$ and obtain a new abstraction M_{α_2} (cf. Figure 2e). Then, in the second iteration of the refinement loop, we will run the loop body on M_{α_2} (cf. Figure 2f), obtaining M'_{α_2}. Running $\texttt{apply}(M_{\alpha_2}, M'_{\alpha_2}, \texttt{refine}[\tau_2, \sigma_2, \alpha_2])$ will not find any inconsistency this time ($\tau_2 = \{a \mapsto a, b \mapsto c\omega, c \mapsto b\omega\}$), so we can terminate the refinement. Applying M'_{α_2} on M_q with τ_2 once, we obtain the tree in Figure 2g. □

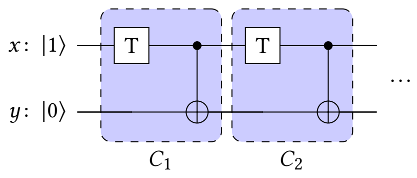


Figure 3: An example circuit for loop summarization

Formally, the algorithm computes a *summary* for a sequence of gates C w.r.t. a quantum state q (represented by an MTBDD M_q). The summary is a pair (M_α, τ) where M_α is a stable abstraction of M_q (w.r.t. C) and τ denotes how the symbolic variables should be updated during one loop iteration, computed as follows. On Line 2, we perform the initial abstraction of M_q, obtaining an MTBDD $M_\alpha^{\mathit{refined}}$ with one symbolic variable from $\mathbb{S}$ (the set of symbolic variables) for every amplitude occurring in M_q (the mapping is remembered in α). Then, we execute the sequence of gates C over M_q obtaining M'_q, where the resulting amplitudes are represented by symbolic terms over $\mathbb{S}$ (Line 5). On Line 8, we collect into τ the information about how the symbolic variables were updated and check whether all bases mapping to the same symbolic variable are updated in the same way—if not (on Line 19, we emphasize that we do not just check the *identity* of the two symbolic terms but, instead, check their *semantic equivalence*), we refine the abstraction (by introducing new symbolic variables for bases that have a different update) and try again. When we reach the fixpoint, we return the resulting abstracted MTBDD M_α together with the updates τ.

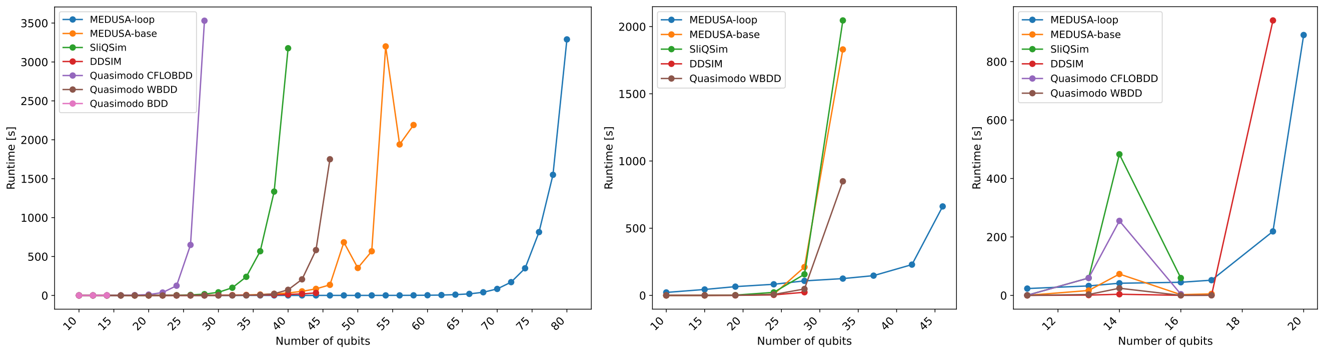
5 IMPLEMENTATION

We implemented the proposed techniques in a prototype called Medusa [22]. Medusa is written in C and uses Sylvan [31] for handling MTBDDs and the GNU GMP library [1] for handling integers of arbitrary length. We use two configurations of Medusa: with (Medusa$_{loop}$) and without (Medusa$_{base}$) loop summarization.

To achieve accuracy, we represent complex numbers algebraically as proposed in [37] and first realized in [30] (used also later in [11, 12]). The algebraic representation is given by the form

$$\left(\frac{1}{\sqrt{2}}\right)^k (a + b\omega + c\,\omega^2 + d\,\omega^3), \tag{1}$$

where a, b, c, d, and k are integers. A complex number is then represented by a five-tuple (a, b, c, d, k). Although it only represents a countable subset of $\mathbb{C}$, it can approximate any complex number up to a specified precision and suffices to support a set of quantum gates for universal quantum computation. The algebraic representation also allows for efficient encoding of some operations. For example, because $\omega^4 = -1$, the multiplication of (a, b, c, d, k) by ω can be carried out by a simple right circular shift of the first four entries and then taking the opposite number for the first entry, namely $(-d, a, b, c, k)$, which represents the complex number $\left(\frac{1}{\sqrt{2}}\right)^k(-d + a\omega + b\omega^2 + c\,\omega^3)$.



(a) Grover's search (b) Period finding (c) Quantum counting

Figure 4: Runtimes of the simulators on the LOOPS benchmark.

6 EXPERIMENTAL RESULTS

Simulators. We compared the performance of MEDUSA against the following state-of-the-art quantum circuit simulators: SLIQSIM [30], QUASIMODO [28], DDSIM [38] (v1.21.0), and QUOKKA# [24]. For QUASIMODO, which contains 3 different backends (BDD, WBDD, and CFLOBDD), we use QUAS[B] to denote the version that uses backend B (we note that its WBDD backend uses a decision diagram package from DDSIM). To the best of our knowledge, only MEDUSA and SLIQSIM perform *accurate* simulation (using algebraic encoding of complex numbers) while the other tools use floating-point numbers (with possible numerical errors). The importance of accurate simulation has been demonstrated in applications such as quantum circuit equivalence checking [34]. All experiments were conducted on a server with two Intel Xeon X5650 (2.67 GHz) CPUs, 32 GiB of RAM running Debian GNU/Linux 12, with the timeout of 60 min.

Benchmarks. We performed experiments on the following two benchmark sets of quantum circuits in OPENQASM:

- LOOPS: This benchmark set contains circuits containing loops with fixed numbers of iterations. The particular circuits are implementations of Grover's search algorithm [19] (with a single solution), quantum counting [8], and period finding [23], the last two without the final inverse quantum Fourier transform (QFT)[2]. For quantum counting and period finding, we created several families of circuits with increasing size, denoted as $\langle FR\rangle\_\langle SR\rangle\_\langle MT\rangle$, where FR denotes the number of qubits in the *first register*, SR denotes the number of qubits in the *second register* (cf. [8]), and MT denotes the number of randomly generated multi-control Toffoli gates in the oracle. We always set $SR = \left\lfloor \frac{FR}{2} \right\rfloor$ and $MT \in \{5, 10, 15\}$. We unfolded the loops for tools that did not support them.
- STRAIGHTLINE: This benchmark set contains circuits without loops implementing Bernstein-Vazirani's algorithm [6] (from 2 to 100 qubits $\rightsquigarrow$ 99 circuits), multi-control Toffoli gates (from 6 to 198 qubits with a step of 2 $\rightsquigarrow$ 97 circuits), benchmarks from the toolkit FEYNMAN [3] (43 circuits), multi-oracle version of Grover's search (without loops; 9 circuits; MOG) from [2], randomly generated circuits from [2] (97 circuits), REVLIB benchmarks [35] (80 circuits), and modifications of certain REVLIB benchmarks from [30] (16 circuits) denoted as REVLIB-H (these were obtained by inserting an H gate at each unassigned input).

[2]We did not include the inverse QFT because it requires rotations by $\frac{\pi}{2^n}$ for arbitrary n, which are not supported by our prototype, since it uses the algebraic encoding of complex numbers from Sec. 5. Note that this is not a conceptual limitation; one could solve it precisely by, e.g., dynamically refining the algebraic encoding to use finer base rotation than $\frac{\pi}{4}$, in particular $\frac{\pi}{2^n}$, or, not preserving accuracy, one could convert the algebraic encoding into floating-point numbers and continue with them. We wish to develop such solutions in our future work.

The experiments measured the time it took for the final quantum state to be obtained in the given representation except QUOKKA#, where we measured the time to obtain the probability of the first qubit being zero (QUOKKA# does not compute representations of quantum states). The benchmarks did not contain measurements. A reproduction package for the experiments is available at [10].

Research questions. We were interested in the following two key research questions related to the proposed approach.

RQ1 What is the impact of loop summarization on the performance of quantum simulators?

RQ2 How does the MTBDD-based representation with custom gate operations compare to other simulators?

RQ1: Loop Summarization

For answering the first research question, which is the main target of this paper, we ran the simulators on the LOOPS benchmark set. The results can be seen in Figure 4 (for period finding and quantum counting, we show results for the families of circuits with oracle composed of 5 random multi-control Toffoli gates). Moreover, in Table 1, we give selected concrete results (we included for every simulator the largest circuit in the family where it finished). QUOKKA# is not included since it did not finish on any of the circuits. We also encountered some issues when running QUAS[CFLOBDD] (internal error) and QUAS[BDD] (incorrect implementation of the multi-control Toffoli `mcx` gate), which are labelled as ERR.

We first focus on comparing the performance of MEDUSA$_{loop}$ and MEDUSA$_{base}$, which differ only in loop summarization. The results show that in all three algorithms, MEDUSA$_{loop}$ scales much better than MEDUSA$_{base}$—it manages to simulate circuits of a size (the number of gates) one to three orders of magnitude larger. According to the results, the amount of necessary computation is significantly decreased, so we believe we can expect a similar behaviour if loop

Table 1: Results for the Loops benchmark set (for every family, we include circuits which were the largest ones that some of the simulators managed to simulate before timeout). The columns "#q" and "#G" denote the number of qubits and gates (after loop unrolling) respectively. Times are given in seconds ("0" denotes a time <0.5 s), memory in MiB. TO denotes a timeout, ERR denotes an error, num denotes the fastest time, and num denotes the fastest *accurate* simulator (Medusa or SliQSim).

				Medusa$_{loop}$		Medusa$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]	
	circuit	**#q**	**#G**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**
Grover	7	14	480	0	99	0	37	0	12	0	30	0	463	0	444	1	445
	11	22	3,337	0	122	0	42	1	12	0	34	37	774	0	450	TO	TO
	14	28	12,115	0	145	1	56	17	13	1	50	3,530	9,532	0	470	TO	TO
	20	40	140,721	0	187	32	387	3,176	25	12	118	TO	TO	73	769	TO	TO
	22	44	310,367	0	196	85	1,088	TO	TO	32	254	TO	TO	583	1,083	TO	TO
	23	46	461,646	0	200	136	1,735	TO	TO	TO	TO	TO	TO	1,750	1,708	TO	TO
	29	58	4,676,916	2	214	2,190	10,032	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	40	80	292,359,936	3,290	251	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Period Finding	16_08_05	24	1,507,322	83	600	8	24	23	130	4	1,235	ERR	ERR	7	449	ERR	ERR
	19_09_15	28	39,321,545	109	2,154	247	32	587	3,002	178	31,144	ERR	ERR	198	452	ERR	ERR
	22_11_05	33	146,800,628	125	922	1,830	38	2,046	10,293	TO	TO	ERR	ERR	849	454	ERR	ERR
	22_11_15	33	448,790,444	128	1,662	3,020	27	TO	TO	TO	TO	ERR	ERR	2,650	454	ERR	ERR
	31_15_15	46	277,025,390,495	673	1,973	TO	TO	TO	TO	TO	TO	ERR	ERR	TO	TO	ERR	ERR
Counting	10_05_05	16	40,937	45	2,115	3	83	60	15	0	42	4	459	0	446	ERR	ERR
	11_05_05	17	81,898	52	2,116	5	109	TO	TO	0	65	TO	TO	0	447	ERR	ERR
	12_06_15	19	376,760	250	7,691	TO	TO	TO	TO	1,280	294	TO	TO	TO	TO	ERR	ERR
	13_06_15	20	753,593	919	9,502	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR

summarization is implemented for other representations. Therefore, the answer to RQ1 is that *the impact of loop summarization is profound for the performance of the simulator on circuits with loops.*

Let us also compare the performance with the other simulators in this benchmark set. We can see that in the case of Grover's algorithm (Figure 4a), Medusa$_{loop}$ managed to verify instances of a size far beyond the capabilities of any other simulator, in particular 80 qubits. The second best-performing simulator was Medusa$_{base}$, which scaled up to 58 qubits, followed by Quas[WBDD] (46 qubits), DDSIM (44 qubits), and SliQSim (40 qubits). The situation is similar for period finding (Figure 4b), where Medusa$_{loop}$ can scale up to 46 qubits, while the second best ones, Quas[WBDD] and Medusa$_{base}$, can scale only to 33 qubits. Let us note the size of the largest period finding circuit that Medusa$_{loop}$ managed to simulate in 12 minutes: over 277 billion gates. To the best of our knowledge, no existing quantum simulator is able to scale up to circuits of this size. Similar situation repeats for quantum counting, Medusa$_{loop}$ can, again, scale up to circuits of complexities that no other simulator could handle (although, due to the complexity of the circuits, it does not perform so well on smaller-sized circuits).

RQ2: MTBDD-Based Simulator

To answer the second research question, in addition to the results from the Loops benchmark set, we also evaluated the performance of simulators on the StraightLine benchmark (these circuits did not use loops, so we do not include Medusa$_{loop}$, since it would be the same as Medusa$_{base}$). Due to space limitations, we present only selected results. We chose circuits that took over one second to finish for three better-performing tools Medusa$_{base}$, SliQSim, and DDSIM. However, RevLib-H circuits were challenging for most tools, except for SliQSim which solved 13 cases. Both Medusa$_{base}$ and DDSIM solved 5 cases in RevLib-H. SliQSim splits amplitude values into bits and uses multiple BDDs to store a quantum state, resulting in better compression in this benchmark. Instead of showing a large table filled with TO, we show only the 5 solved cases in RevLib-H and refer readers to [30] for a more extensive comparison of SliQSim and DDSIM. Note that some tools had issues on some of the benchmarks due to unsupported gates.

The results show that Medusa$_{base}$ is competitive to other simulators and in many cases, especially for the challenging benchmarks from Feynman, is the best available accurate simulator. For the Loops benchmark, as mentioned previously, Medusa$_{base}$ is performing well also compared to other simulators: it is the best one on Grover and performs well also on the other two (it beats SliQSim, the only other accurate simulator). To conclude, the answer to RQ2 is that *the MTBDD-based representation with custom gate operations is competitive to other simulators, often complementary to SliQSim.*

7 RELATED WORK

DDSIM [38] is a quantum circuit simulator based on *quantum multiple-valued decision diagrams* (QMDDs) [25], which support representation and multiplication of state vectors and operator matrices. In [21], a QMDD variant, called *tensor decision diagrams* (TDDs), is proposed to allow tensor-network-like quantum circuit simulation. The TDD performance is comparable to DDSIM [21].

SliQSim [30] exploits the standard *reduced ordered binary decision diagrams* (ROBDDs) [9] to represent quantum states exactly with an algebraic number system and achieves precise quantum operations through Boolean formula manipulation. Note that similarly to Medusa, the supported quantum gate set of SliQSim, though universal, is restricted to those algebraically representable.

The paper [12] proposes verification of quantum circuits using *tree automata* to model their pre- and post-conditions. This method helps create an automatic verification framework that checks the correctness of the quantum circuit against a user-specified specification. Tree automata, similarly to decision diagrams, can efficiently represent identical subtrees using the same structure. Furthermore, they can use non-deterministic choice to represent multiple states in the same structure. We took inspiration from their extension to symbolic amplitudes in [11] to develop our symbolic execution.

Table 2: Selection of results for the StraightLine benchmark. The columns "#q" and "#G" denote the number of qubits and gates respectively. Times are given in seconds ("0.00" denotes a time <0.01 s), memory in MiB. TO denotes a timeout, ERR denotes an error, num denotes the fastest time, and num denotes the fastest *accurate* simulator (Medusa or SliQSim). We do not mark Quokka# as the fastest because it does not compute the quantum state representation.

				Medusa$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]		Quokka#	
	circuit	#q	#G	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
Feynman	gf2^{32}_mult	96	3,322	0.26	39	1.34	12	0.10	70	0.72	459	0.11	501	0.91	449	0.86	45
	gf2^{64}_mult	192	12,731	1.82	65	17.11	19	0.74	126	2.76	463	0.68	600	4.35	461	3.56	148
	gf2^{128}_mult	384	50,043	20.40	231	264.81	37	5.28	234	10.70	477	4.76	1,158	27.40	498	15.39	570
	gf2^{256}_mult	768	198,395	163.00	1,634	TO	TO	41.21	538	42.50	531	38.50	4,988	231.00	632	71.28	2,324
	hwb8	12	6,446	0.16	38	3.69	12	0.03	33	1.04	460	0.03	443	1.09	443	TO	TO
	hwb10	16	31,764	0.79	50	84.20	15	0.21	38	4.74	465	0.22	447	1.70	445	TO	TO
	hwb11	15	87,789	2.64	103	660.92	22	0.49	70	12.70	474	0.51	448	1.59	448	TO	TO
	hwb12	20	171,482	5.80	204	2,568.02	34	1.13	132	26.90	509	1.35	455	6.48	457	3,193.78	1,069
MOG	10	30	2,433	0.20	41	1.25	12	0.07	34	9.50	594	0.05	456	TO	TO	62.68	40
	11	33	3,746	0.36	44	3.12	12	0.12	42	52.00	905	0.08	462	TO	TO	167.00	56
Random	85	85	255	0.99	51	0.46	14	2.11	63	ERR	ERR	0.10	485	ERR	ERR	0.03	12
	86	86	258	15.30	213	0.47	14	2.24	72	ERR	ERR	3.25	553	ERR	ERR	0.07	12
	89	89	267	9.48	105	0.67	14	0.72	65	ERR	ERR	0.59	491	ERR	ERR	0.06	12
	93	93	279	1.68	61	0.32	13	0.18	67	ERR	ERR	0.10	493	ERR	ERR	0.04	12
	94	94	282	79.60	337	0.77	17	4.45	76	ERR	ERR	74.30	521	ERR	ERR	0.07	12
	97	97	291	5.70	117	0.42	13	1.46	77	ERR	ERR	0.42	524	ERR	ERR	0.03	12
	99	99	297	9.58	173	0.38	12	2.61	78	ERR	ERR	0.67	525	ERR	ERR	0.08	12
RevLib	apex5_290	1,025	2,909	1.75	61	0.37	43	1.02	535	0.30	466	1.33	1,214	4.16	516	2.10	72
	cps_292	923	2,763	1.19	57	0.20	30	1.25	484	0.24	464	1.09	1,035	2.99	527	1.38	59
	frg2_297	1,219	3,724	2.32	93	0.49	48	1.51	633	0.36	468	1.90	1,307	6.32	497	2.15	84
	seq_314	1,617	5,990	4.96	97	1.35	108	4.11	834	0.62	476	3.71	1,775	13.90	536	3.65	124
RevLib-H	add64_184	193	385	0.19	203	0.02	13	0.09	117	ERR	ERR	0.07	545	ERR	ERR	ERR	ERR
	cpu_register_32_405	328	890	0.46	213	0.08	14	0.41	194	ERR	ERR	0.70	668	ERR	ERR	ERR	ERR
	e64-bdd_295	195	452	1.98	238	2.48	13	2.00	126	0.65	476	0.54	613	ERR	ERR	ERR	ERR
	ex5p_296	206	655	7.61	283	12.02	21	3.56	132	ERR	ERR	1.15	691	ERR	ERR	ERR	ERR
	hwb9_304	170	708	33.00	662	13.50	20	12.16	114	ERR	ERR	4.90	1,105	ERR	ERR	ERR	ERR

SymQV [5] encodes quantum circuit verification problems into SMT with the theory of real numbers, using variables in trigonometric functions, which might lose precision in corner cases. Their approach requires 2^n variables to encode a n-qubit circuit in the worst case. A polynomial SMT encoding of quantum circuits was introduced in [13], where an extension of array theory, named *the theory of cartesian arrays (CaAL)*, was proposed and used to encode quantum gates. Both methods are effective only for small circuits.

Quasimodo [28] is a simulation tool with multiple backends, including BDDs, weighted BDDs (using the backend of DDSIM), and *context-free language ordered binary decision diagrams* (CFLOBDDs) [29], which combine BDDs with pushdown automata.

Hong *et al.* [20] proposed *symbolic TDDs* (symTDDs) for symbolically executing and representing quantum circuits and quantum states. However, in quantum circuit simulation, parameters are typically predetermined, making this approach useful mainly for parameterized quantum circuit equivalence checking.

Quokka# [24] extended the standard stabilizer formalism [17] to present a general pure state using its stabilizers. The representation circumvents complex numbers and only requires manipulating weights in real (possibly negative) numbers for the supported quantum gate operations. Thereby, quantum circuit simulation can be encoded into a weighted model counting problem. Quokka# only supports Clifford+T and rotation gates (which is, however, universal). Experimental results show the advantages of Quokka# on certain benchmarks such as quantum Fourier transform (QFT).

Although Clifford circuits should be efficiently simulatable according to the Gottesman–Knill theorem [18], simulating them in decision diagrams may suffer from exponential growth in size. To overcome this problem, Vinkhuijzen *et al.* [32, 33] proposed the *local invertible map decision diagrams* (LIMDDs), a data structure based on QMDDs that further merges nodes that are equivalent up to a *local invertible map* (LIM). LIMDDs successfully combine decision diagrams and the stabilizer formalism, and they efficiently overcome the challenge of exponential growth in decision diagrams on Clifford circuits. The authors of [32, 33] demonstrated that LIMDDs are more scalable in simulating QFT circuits than QMDDs.

8 CONCLUSION

We presented a technique for accelerating the simulation of quantum circuits with loops by computing the loops' summaries using symbolic execution. The experiments show that this technique enables the simulation of quantum circuits previously believed to be infeasible. In the future, we wish to further develop the loop summarization by integrating it with other data structures. Moreover, we wish to look at the problem of automatically generalizing a computed summary into a *closed form* (such as the description "$K\omega\,|11\rangle$ if K is odd and $K\omega\,|10\rangle$ if K is even" from Example 1), and use the technique also in the verification framework of [12].

ACKNOWLEDGMENTS

This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, the FIT BUT internal project FIT-S-23-8151, the National Science and Technology Council of Taiwan under grant 113-2119-M-002-024, and the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903.

REFERENCES

[1] 2022. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/

[2] 2024. The AutoQ repository. https://github.com/alan23273850/AutoQ/

[3] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS)*, Peter Selinger and Giulio Chiribella (Eds.), Vol. 287. 1–21. https://doi.org/10.4204/EPTCS.287.1

[4] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, et al. 1997. Algebraic Decision Diagrams and Their Applications. *FMSD* 10, 2/3 (1997), 171–206. https://doi.org/10.1023/A:1008699807402

[5] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. symQV: Automated Symbolic Verification of Quantum Programs. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.), Vol. 14000. Springer, 181–198. https://doi.org/10.1007/978-3-031-27481-7_12

[6] Ethan Bernstein and Umesh V. Vazirani. 1993. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 11–20. https://doi.org/10.1145/167088.167097

[7] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. In *Quantum computation and information (Washington, DC, 2000)*. Contemp. Math., Vol. 305. Amer. Math. Soc., Providence, RI, 53–74. https://doi.org/10.1090/conm/305/05215

[8] Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Quantum Counting. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings (Lecture Notes in Computer Science)*, Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel (Eds.), Vol. 1443. Springer, 820–831. https://doi.org/10.1007/BFB0055105

[9] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[10] Tian-Fu Chen, Yu-Fang Chen, Jie-Hong Jiang, Sára Jobranová, and Ondřej Lengál. 2024. Artifact for the ICCAD'24 paper "Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization". https://doi.org/10.5281/zenodo.13243595

[11] Yu-Fang Chen, Kai-Min Chung, Ondrej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. 2023. AutoQ: An Automata-Based Quantum Circuit Verifier. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13966. Springer, 139–153. https://doi.org/10.1007/978-3-031-37709-9_7

[12] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM Program. Lang.* 7, PLDI, Article 156 (jun 2023), 26 pages. https://doi.org/10.1145/3591270

[13] Yu-Fang Chen, Philipp Rümmer, and Wei-Lun Tsai. 2023. A Theory of Cartesian Arrays (with Applications in Quantum Circuit Verification). In *International Conference on Automated Deduction*. Springer, 170–189.

[14] Bob Coecke and Ross Duncan. 2008. Interacting Quantum Observables. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.), Vol. 5126. Springer, 298–310. https://doi.org/10.1007/978-3-540-70583-3_25

[15] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (sep 2022), 50 pages. https://doi.org/10.1145/3505636

[16] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods Syst. Des.* 10, 2/3 (1997), 149–169. https://doi.org/10.1023/A:1008647823331

[17] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction*. Ph.D. Dissertation. California Institute of Technology.

[18] Daniel Gottesman. 1998. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006* (1998).

[19] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 212–219. https://doi.org/10.1145/237814.237866

[20] X. Hong, W. Huang, W. Chien, Y. Feng, M. Hsieh, S. Li, C. Yeh, and M. Ying. 2023. Decision Diagrams for Symbolic Verification of Quantum Circuits. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 970–977. https://doi.org/10.1109/QCE57702.2023.00111

[21] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. 2022. A Tensor Network based Decision Diagram for Representation of Quantum Circuits. *ACM Trans. Des. Autom. Electron. Syst.* 27, 6, Article 60 (jun 2022), 30 pages. https://doi.org/10.1145/3514355

[22] Sára Jobranová. 2024. The Medusa repository. https://github.com/s-jobra/MEDUSA/

[23] Alexei Y. Kitaev. 1996. Quantum measurements and the Abelian Stabilizer Problem. *Electron. Colloquium Comput. Complex.* TR96-003 (1996). ECCC:TR96-003 https://eccc.weizmann.ac.il/eccc-reports/1996/TR96-003/index.html

[24] Jingyi Mei, Marcello Bonsangue, and Alfons Laarman. 2024. Simulating Quantum Circuits by Model Counting. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, Canada, July 22-27, 2024, Proceedings, Part III (Lecture Notes in Computer Science)*, Arie Gurfinkel and Vijay Ganesh (Eds.), Vol. 14683. Springer, 555–578.

[25] Philipp Niemann, Robert Wille, D. Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. 2016. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35, 1 (2016), 86–99. https://doi.org/10.1109/TCAD.2015.2459034

[26] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Nocon, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. 2019. Massively parallel quantum computer simulator, eleven years later. *Comput. Phys. Commun.* 237 (2019), 47–61. https://doi.org/10.1016/J.CPC.2018.11.005

[27] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

[28] Meghana Sistla, Swarat Chaudhuri, and Thomas W. Reps. 2023. Symbolic Quantum Simulation with Quasimodo. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13966. Springer, 213–225. https://doi.org/10.1007/978-3-031-37709-9_11

[29] Meghana Aparna Sistla, Swarat Chaudhuri, and Thomas Reps. 2023. CFLOBDDs: Context-free-language ordered binary decision diagrams. *ACM Transactions on Programming Languages and Systems* (2023).

[30] Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. 2021. Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 439–444. https://doi.org/10.1109/DAC18074.2021.9586191

[31] Tom van Dijk and Jaco van de Pol. 2017. Sylvan: multi-core framework for decision diagrams. *Int. J. Softw. Tools Technol. Transf.* 19, 6 (2017), 675–696. https://doi.org/10.1007/S10009-016-0433-2

[32] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. 2023. LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States. *Quantum* 7 (2023), 1108. https://doi.org/10.22331/Q-2023-09-11-1108

[33] Lieuwe Vinkhuijzen, Thomas Grurl, Stefan Hillmich, Sebastiaan Brand, Robert Wille, and Alfons Laarman. 2023. Efficient Implementation of LIMDDs for Quantum Circuit Simulation. In *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings (Lecture Notes in Computer Science)*, Georgiana Caltais and Christian Schilling (Eds.), Vol. 13872. Springer, 3–21. https://doi.org/10.1007/978-3-031-32157-3_1

[34] Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R. Jiang. 2022. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, New York, NY, USA, 523–528. https://doi.org/10.1145/3489517.3530481

[35] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *38th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2008), 22-23 May 2008, Dallas, Texas, USA*. IEEE Computer Society, 220–225. https://doi.org/10.1109/ISMVL.2008.43

[36] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942057

[37] Alwin Zulehner, Philipp Niemann, Rolf Drechsler, and Robert Wille. 2019. Accuracy and Compactness in Decision Diagrams for Quantum Computation. In *2019 Design, Automation and Test in Europe Conference (DATE)*. 280–283. https://doi.org/10.23919/DATE.2019.8715040

[38] Alwin Zulehner and Robert Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427