BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EFFICIENT AUTOMATA TECHNIQUES AND THEIR APPLICATIONS

EFEKTIVNÍ AUTOMATOVÉ TECHNIKY A JEJICH APLIKACE

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — **Ing. VOJTĚCH HAVLENA**
AUTOR PRÁCE

SUPERVISORS — **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL

CO-SUPERVISOR — **Ing. ONDŘEJ LENGÁL, Ph.D.**
ŠKOLITEL SPECIALISTA

BRNO 2021

Abstract

This thesis develops efficient techniques for finite automata and their applications. In particular, we focus on finite automata in network intrusion detection and automata in decision procedures and verification. In the first part of the thesis, we propose techniques of approximate reduction of nondeterministic automata decreasing consumption of resources of hardware-accelerated deep packet inspection. The second part is devoted to automata in decision procedures, in particular, to weak monadic second-order logic of k successors (WSkS) and the theory of strings. We propose a novel decision procedure for WS2S based on automata terms allowing one to effectively prune the state space. Further, we study techniques of WSkS formulae preprocessing intended to reduce the sizes of constructed intermediate automata. Moreover, we employ automata in a decision procedure of the theory of strings for efficient handling of the proof graph. The last part of the thesis then proposes optimizations in rank-based Büchi automata complementation reducing the number of generated states during the construction.

Abstrakt

Tato práce se zabývá vývojem efektivních technik pro konečné automaty a jejich aplikace. Zejména se věnujeme konečným automatům použitých pří detekci útoků v síťovém provozu a automatům v rozhodovacíh procedurách a verifikaci. V první části práce navrhujeme techniky přibližné redukce nedeterministických automatů, které snižují spotřebu zdrojů v hardwarově akcelerovaném zkoumání obsahu paketů. Druhá část práce je je věnována automatům v rozhodovacích procedurách, zejména slabé monadické logice druhého řádů k následníků (WSkS) a teorie nad řetězci. Navrhujeme novou rozhodovací proceduru pro WS2S založenou na automatových termech, umožňující efektivně prořezávat stavový prostor. Dále studujeme techniky předzpracování WSkS formulí za účelem snížení velikosti konstruovaných automatů. Automaty jsme také aplikovali v rozhodovací proceduře teorie nad řetězci pro efektivní reprezentaci důkazového stromu. V poslední části práce potom navrhujeme optimalizace rank-based komplementace Büchiho automatů, které snižuje počet generovaných stavů během konstrukce komplementu.

Keywords

Finite automata, approximate reduction, minimization, tree automata, decision procedures, WSkS, automata terms, antiprenexing, theory of strings, quadratic word equations, Büchi automata complementation, rank-based complementation

Klíčová slova

Konečné automaty, přibližná redukce, minimalizace, stromové automaty, rozhodovací procedury, WSkS, automatové termy, antiprenexní forma, teorie řetězců, kvadratické rovnice nad slovy, komplementace Büchiho automatů, rank-based komplementace

Reference

HAVLENA, Vojtěch. *Efficient Automata Techniques and Their Applications* . Brno, 2021. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisors prof. Ing. Tomáš Vojnar, Ph.D., Ing. Ondřej Lengál, Ph.D.

Efficient Automata Techniques and Their Applications

Declaration

I hereby declare that this PhD thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. and Ing. Ondřej Lengál, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Havlena
June 28, 2021

Acknowledgements

First of all, I want to thank my supervisors Tomáš Vojnar and Ondra Lengál for their support, patience, and help during my studies. It was a great ride. Special thanks belong to Ondra Lengál. He taught me a lot not just about science, but also about life. I feel I was pretty lucky to have such supervisors. Also, I would like to thank my co-authors (listed in the alphabetical order), in particular, Milan Češka, Yu-Fang Chen, Lukáš Holík, Jakub Semrič, Andrea Turrini, Ondřej Valeš, and the guys from ANT@FIT for their collaboration. I thank also all people from the VeriFIT group for making the study more pleasant. I thank Yu-Fang Chen for hosting me in Academia Sinica. Last but not least I want to thank my family and friends for their neverending support.

The work presented in this thesis was supported by the Ministry of Education, Youth and Sports Czech Republic (under the projects LL1908, LQ1602), the Czech Science Foundation (under the projects GA16-17538S, GJ16-24707Y, GA17-12465S, GA19-24397S, GA20-07487S, GJ20-02328Y), the internal BUT FIT projects FIT-S-17-4014, FIT-S-20-6427.

Contents

Chapter 1

Introduction

The finite automaton is a simple and powerful model of computation. Since its inception in the 40s, when a notion similar to a finite automaton was proposed in the context of neurophysiology [209], through the works of Mealy, Moore, Rabin, Scott [236, 216, 210], and others in the 50s to this date, it constantly attracts the attention of researchers and gained a position of tremendous importance in computer science. During the years there emerged different automata models extending the original model, such as automata with counters, registers, probabilities, or automata over finite/infinite trees, pushing the scope of finite automata even further. The ingenuity of the simple model of finite automata has been witnessed by long series of real-world applications across computer science. Since a lot of systems have a finite state nature, finite automata are an important tool for their description and reasoning about them. Finite automata touch almost every field of computer science, in particular software/hardware verification, bug hunting, parsing (lexical analysis within compilers), decision procedures for various logics, regular expression matching, bioinformatics, or speech recognition. Efficient handling of finite automata is hence a crucial task. However, many problems related to finite automata are inherently hard. For instance, inclusion checking of two nondeterministic finite automata (NFAs) is a **PSPACE**-complete problem, the same as for their state-based minimization. Heuristics and efficient techniques for problems involving finite automata are hence desirable. In this thesis, we focus on efficient techniques concerning finite automata in a couple of current applications mentioned below, in particular, hardware-accelerated intrusion detection, decision procedures for various logics, and verification.

The first studied topic covers reduction of NFAs in *hardware-accelerated intrusion detection.* The ever-increasing speed of computer networks brings new security challenges as well as opportunities for malicious users. In order to identify malicious traffic, network intrusion detection systems monitoring incoming packets are often deployed. Suspicious packets, i.e., packets that may belong to an attack or other malicious activity, can be specified not only by IP addresses or ports, but also by regular expressions describing potentially dangerous packet content. Intrusion detection systems then perform matching of payloads of incoming packets, also called the *deep packet inspection*, against the regular expressions of interest. A problem arises especially when the deep packet inspection is applied on high-speed networks (i.e., networks with the speed of 100 Gbps and beyond). In such a case, a single-box software solution cannot achieve sufficient throughput and hence either clusters of many computers and/or specialized hardware units accelerating regular expression matching are necessary. Hardware matching engines can directly implement nondeterministic finite automata obtained from regular expressions [254]. The hardware

resources required for storage of these finite automata are, however, very restricted (on top of that, pattern matching on high-speed networks may need to implement multiple copies of the matching unit containing the finite automaton [202]). Therefore, efficient techniques to reduce NFAs in the context of deep packet inspection, which we address in the first part of the thesis, are much welcome.

The second topic this thesis focus on involves finite automata in decision procedures and verification. Due to a massive spread and a rapid development of computer programs, still more demands on performance, reliability, security, and correctness are put on them. Errors in critical systems, as well as security vulnerabilities, can, in the worst case, cause severe damages, including deaths, injuries, financial losses, or security data leaks. In 1996, the Ariane 5 rocket exploded because of a bug in its control system [124]. More recently, the discovery of software flaws of grounded Boeing 737 MAX prevented it from the return to flight [220]. In 2016 a problematic regular expression caused an outage of the Stack Overflow web service [263]. For these reasons, techniques of automated verification, testing, analysis, and optimization of systems became a hot topic in computer science. Some of these techniques are based on finite automata, so efficient algorithms for handling finite automata are necessary for their scalability.

In software verification, such as approaches based on symbolic execution, correctness of a system is sometimes specified by invariants and assertions that need to hold at given control locations. Invariants and assertions are may be specified by formulae in a suitable decidable logic/theory (or their combination). For instance, for string manipulating programs, a suitable logic can be the first-order *theory of strings*, for integer programs *Presburger arithmetic*, and for heap manipulating programs it can be the *weak monadic second order logic of k successors* (WSkS). The verification process then checks, in cooperation with a logic/theory solver, if all paths in the system (or at least all paths up to some bound) satisfy the assertions and invariants. A combination of suitable logics/theories and efficient solvers may find critical vulnerabilities in various applications, e.g., cross-side-scripting (XSS) vulnerabilities in the context of web applications. Logic/theory solvers are currently bottlenecks of the verification process. In order to verify large scale systems it is hence important to have *efficient* solvers.

However, decision procedures for many logics/theories suffer from a high worst-case complexity. The currently best known algorithm for satisfiability checking of word equations, which is a fundamental stone for a decision procedure for the existential fragment of the theory of strings, runs in a polynomial space [229]. Also notice that the full theory of strings (with unrestricted quantifiers) is undecidable. Moreover, the WSkS logic lies at the edge of decidability, since this logic is decidable with the **NONELEMENTARY** complexity. Efficient decision procedures and heuristics for these logics/theories are hence highly desirable. A promising way for improvements is offered by an efficient employment of finite automata in decision procedures of these logics, which is also a topic of this thesis.

In a more heavyweight approach to formal verification, such as model checking, temporal specifications of the systems being checked may be considered. Such a specification could be provided as a formula in a suitable logic or directly as a finite automaton specifying a correct behavior. If both a model of a system and the specification are given as finite automata $\mathcal{B}$ and $\mathcal{A}$, respectively, the correctness of the system is being checked as $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$, which is equivalent to checking whether $\mathcal{L}(\mathcal{B}) \cap \overline{\mathcal{L}(\mathcal{A})} = \emptyset$. The automaton $\mathcal{B}$ represents the behavior of the system and the automaton $\mathcal{A}$ describes all correct behaviors (directly constructed or obtained, e.g., from a specification provided in some logic, such as LTL, QPTL, or S1S). In the case of verification of reactive systems, both $\mathcal{A}$ and

$\mathcal{B}$ can be encoded by finite automata over infinite words (e.g., Büchi automata). A fundamental stone and bottleneck of efficient language inclusion checking is *complementation*. Complementation for nondeterminitistic automata, particularly, over infinite words, is still a challenging problem despite its intensive study. The existing complementation algorithms still provide a lot of space for improvements because they often suffer from state explosion caused by generating unnecessary states.

1.1 Goals of the Thesis

The main goal of this thesis is to improve the state-of-the-art techniques for efficient automata handling in the context of formal verification and optimization of systems. In particular, we identified three subgoals. In the first subgoal, we aim at an extension of the state-of-the-art approaches for reduction of nondeterministic finite automata over finite words in the context of hardware-accelerated network intrusion detection. Our goal is to propose suitable language-nonpreserving reductions that will be used in addition to the current precise (language preserving) reductions. The second subgoal is to improve decision procedures for the WSkS logic and for fragments of the theory of strings. In particular, we aim at an efficient employment of finite (word/tree) automata in these procedures in order to reduce the size of the generated state space. The third subgoal concerns optimization techniques for Büchi automata complementation, with the focus on rank-based complementation. Despite the fact that currently the best algorithm of rank-based complementation is worst-case optimal, it can still generate a lot of unnecessary states. Therefore, techniques of state space reduction during the construction could improve its practical efficiency.

1.2 Overview of the Achieved Results

In this section, we briefly summarize the main results achieved in this thesis. According to the goals stated above, this thesis is divided into three parts. The first part covers reduction of NFAs in the context of network intrusion detection. The second part is devoted to automata in decision procedures, and the third part deals with optimization of Büchi automata complementation.

Approximate reduction of NFAs. Concerning the first subgoal, we introduce several novel language-nonpreserving (approximate) reductions of NFAs in the context of hardware-accelerated network intrusion detection. In order to squeeze NFAs representing regular expressions of interest into a hardware unit, the proposed reductions allow one to specify the maximum number of states of the target automaton. The reductions are steered either by a probabilistic model of traffic or directly by a multiset of packets representing the input traffic. For the first case, we introduced *approximate reductions with formal guarantees* w.r.t. the traffic model. The reduction is then driven by a metric capturing the error of reduction as the probability that a randomly chosen string (w.r.t. the probabilistic model) is misaccepted. According to this metric, we identify less significant parts of the automaton and then we apply a reduction on these parts (a modification and/or removal of states and transitions of the automaton in order to underapproximate or overapproximate the language). Although the optimal approximate reduction is **PSPACE**-complete, we propose heuristics suitable for NFAs occurring in network intrusion detection systems (NIDSes). The

proposed methods allow to increase the throughput of hardware-accelerated deep packet inspection.

For the case of traffic-sample-driven reduction, we propose *lightweight approximate reductions* allowing to identify the less significant parts of the automaton, where reductions are then applied, directly from the traffic sample. The advantage is the complexity—the reduction of an NFA can be performed in a polynomial time (w.r.t. the size of the NFA and the size of the sample). Using our approaches, we were able to obtain hardware-accelerated NIDSes suitable for high-speed networks, i.e., 100 Gbps and even 400 Gbps networks.

Automata in decision procedures. Concerning the second subgoal, we employ efficient automata techniques in decision procedures of WSkS and the theory of strings. First, we generalize the results of [111] and propose automata terms in the WS2S decision procedure (note that WSkS for $k > 2$ can be encoded into WS2S). Automata terms implicitly represent tree automata and they allow us to construct parts of an automaton corresponding to a formula on-demand, only if it is necessary. Our decision procedure based on automata terms reduces the size of the generated state space using techniques such as subsumption pruning, lazy evaluation, eager-termination, or memoization. We show that our approach can outperform the state-of-the-art MONA tool on certain families of formulae.

Second, we optimize the automata-based decision procedure for WSkS using a static preprocessing of an input formula. Our approach introduces several rewriting rules that are applied on the input formula in order to reduce sizes of the constructed intermediate automata. Our rules include pushing quantifiers deeper into the formula (antiprenexing), distribution of conjunctions, or restructuring conjunctions together with a few of supporting rules. Moreover, we parameterize some of our rules by an estimation of automata sizes corresponding to subformulae occurring within the rules. We obtain a model estimating automata sizes by machine learning techniques (particularly using linear regression). The proposed approach can significantly improve the automata-based decision procedure implemented in MONA.

As a third result, we employ finite automata in deciding a fragment of the theory of strings. We express solving of word equations using Nielsen transformation within the regular model checking framework. We propose an efficient regular representation of the proof graph allowing it to be efficiently handled and to avoid redundancies in it. In order to obtain an efficient decision procedure, we use eager minimization of automata representing proof graphs together with symbolic register transducers representing the transformations. Moreover, we extend the approach also to the theory of strings with length and regular constraints. Our experimental evaluation shows that we are able to outperform state-of-the-art solvers on a set of difficult formulae.

Büchi automata complementation. Results related to the third subgoal concern optimizations of rank-based Büchi automata complementation. We start from Schewe's rank-based construction and propose techniques allowing to remove states and transitions which do not alter the language during the construction of the complemented automaton. In particular, we suggest an approach omitting macrostates with "incompatible" rankings based on direct and delayed simulation, which can be cheaply (compared to the complementation itself) computed from an input BA. Further, we use a notion of super-tight runs to keep only macrostates having an as small rank as possible. We propose methods computing the upper bound for the rank of a macrostate from a deterministic support of the input BA. Our optimizations omitting "unnecessary" macrostates are then combined with a technique

allowing to represent multiple runs with a single representative run. Our experimental evaluation shows that using our techniques we can obtain a competitive complementation algorithm with other state-of-the-art approaches of BA complementation, often performing better than the other approaches.

1.3 Plan of the Thesis

According to the above-stated goals, the thesis is organized into three parts with Chapter 2 serving as preliminaries to the following parts. The first part is devoted to approximate reduction of finite automata for network intrusion detection systems. Chapter 3 serves as an introduction to handling of finite automata in the context of NIDSes. Chapter 4 proposes our approximate reduction of NFAs with formal guarantees and Chapter 5 deals with a lightweight approximate reduction of NFAs. The second part of the thesis deals with automata in decision procedures. Chapter 6 introduces Presburger arithmetic and the WSkS logic with a focus on automata-based decision procedures. In Chapter 7, we propose a novel decision procedure for WS2S based on the notion of automata terms. Chapter 8 deals with preprocessing of WSkS formulae. In Chapter 9, we employ finite automata within a decision procedure for a fragment of the theory of strings. In the last part, we optimize Büchi automata complementation. Chapter 10 serves as a brief introduction to Büchi automata complementation with focus on rank-based complementation. In Chapter 11, we employ simulations to reduce the number of generated states during the complementation and Chapter 12 proposes techniques for removing unnecessary states and a reduction of the maximum rank in order to obtain an as small complement as possible. Finally, Chapter 13 concludes the thesis.

Chapter 2

Preliminaries

In this chapter, we give basic definitions related to formal languages, word/tree automata, and operations over them as they are used in the rest of the thesis. In particular, in Section 2.1, we introduce definitions concerning finite/infinite words, trees, and their languages, in Section 2.2, we define finite automata over finite/infinite words and trees, and in Section 2.3, we give basic Boolean operations for automata over finite words and trees.

2.1 Languages

Sets, relations, functions. We use ω to denote the set $\{0, 1, 2, \dots\}$. An *alphabet* Σ is a finite nonempty set of *symbols*. 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 $\{x \mid \exists! i \in \{1, 2\} : x \in X_i\}$.

Given two binary relations R_1, R_2, we define their *composition* $R_1 \circ R_2 = \{(x, z) \mid \exists y : (x, y) \in R_2 \wedge (y, z) \in R_1\}$. Let $\cong \subseteq X \times X$ be an equivalence on X. We use $[x]_\cong$ to denote the equivalence class of $x \in X$ w.r.t $\cong$ and $X/\cong$ to denote the set $\{[x]_\cong \mid x \in X\}$. The *domain* of a partial function $f : X \to Y$ is the set $\mathrm{dom}(f) = \{x \in X \mid \exists y : x \mapsto y \in f\}$, its *image* is the set $\mathrm{img}(f) = \{y \in Y \mid \exists x : x \mapsto y \in f\}$, and its *restriction* to a set Z is the function $f_{|Z} = f \cap (Z \times Y)$. Furthermore, for a total function $f\colon X \to Y$ and a partial function $h\colon X \to Y$, we use $f \lhd h$ to denote the total function $g\colon X \to Y$ defined as $g(x) = h(x)$ when $h(x)$ is defined and $g(x) = f(x)$ otherwise.

Finite words. A *(finite) word (string)* over Σ is a sequence $w = a_1 \dots a_n$ of symbols from Σ, with ϵ denoting the *empty word*. Given a word $w = a_1 \dots a_n$, we use $|w|$ to denote the length n of w. We use $w_1.w_2$ (and often just $w_1 w_2$) 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 $\Sigma^{\leq \ell}$ denotes the set of all words of length at most ℓ. We abbreviate $\{a\}^*$ as a^* for $a \in \Sigma$. A *language* of (finite) words over Σ is a subset L of Σ^*. A *concatenation* of two languages L_1 and L_2, denoted as $L_1.L_2$, is defined as $L_1.L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. To simplify the notation, we sometimes mix concatenation of words and languages with obvious semantics, e.g., $w.L$ for a word w and a language L to denote $\{wu \mid u \in L\}$.

Ordered trees. In this thesis, we will consider ordered k-ary trees. We call a word $p \in \{\mathtt{1}, \dots, \mathtt{k}\}^*$ a tree *position*, and, for each $\mathtt{i} \in \{\mathtt{1}, \dots, \mathtt{k}\}$, we call $p.\mathtt{i}$ its i-th *child*. Given an alphabet Σ s.t. $\bot \notin \Sigma$, a *tree* over Σ is a finite partial function $\tau : \{\mathtt{1}, \dots, \mathtt{k}\}^* \to (\Sigma \cup \{\bot\})$ such that (i) $\mathrm{dom}(\tau)$ is non-empty and prefix-closed, and (ii) for all positions $p \in \mathrm{dom}(\tau)$,

either $\tau(p) \in \Sigma$ and p has all k children, or $\tau(p) = \bot$ and p has no children, in which case it is called a *leaf*. The position ϵ is called the *root*. We write $\Sigma^\star$ to denote the set of all trees over Σ. We abbreviate $\{a\}^\star$ as $a^\star$ for $a \in \Sigma$. A *language* of trees over Σ is a subset L of $\Sigma^\star$.

Infinite words. Let Σ be an alphabet. 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 $\alpha = \alpha_0\alpha_1\ldots$ The suffix $\alpha_i\alpha_{i+1}\ldots$ of α is denoted by $\alpha_{i:\omega}$. We use Σ^ω to denote the set of all infinite words over Σ. A *language* of (infinite) words over Σ is a subset L of Σ^ω. If it is clear from the context, we do not explicitly specify finiteness/infiniteness of words and we speak simply about languages.

2.2 Finite Automata

Finite word automata. A *nondeterministic finite automaton* (NFA) is a quadruple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to 2^Q$ 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 $q' \in \delta(q, a)$. We extend δ to a set of states S as $\delta(S, a) = \bigcup_{q\in S} \delta(q, a)$. Sometimes, we abuse notation and treat δ as a subset of $Q \times \Sigma \times Q$. A sequence of states $\rho = q_0 \cdots q_n$ is a *run* of $\mathcal{A}$ 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 $\mathcal{L}_\mathcal{A}(q) = \{w \mid \exists q_F \in F : q \overset{w}{\rightsquigarrow} q_F\}$. Moreover, we define $\mathcal{L}_\mathcal{A}(p, q) = \{w \mid p \overset{w}{\rightsquigarrow} q\}$. The *banguage* (backward language) of a state q is defined as $\mathcal{L}^\flat_\mathcal{A}(q) = \{w \mid \exists q_I \in I : q_I \overset{w}{\rightsquigarrow} q\}$. Both the language of a state and the banguage can be naturally extended to a set $S \subseteq Q$ as $\mathcal{L}_\mathcal{A}(S) = \bigcup_{q\in S} \mathcal{L}_\mathcal{A}(q)$ and $\mathcal{L}^\flat_\mathcal{A}(S) = \bigcup_{q\in S} \mathcal{L}^\flat_\mathcal{A}(q)$, respectively. We drop the subscript $\mathcal{A}$ when the context is obvious. $\mathcal{A}$ *accepts* the language $\mathcal{L}(\mathcal{A})$ defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(I)$. $\mathcal{A}$ is called *deterministic* (DFA) if $|I| = 1$ and $\forall q \in Q$ and $\forall a \in \Sigma : |\delta(q, a)| \leq 1$ and *complete* if $\forall q \in Q$ and $\forall a \in \Sigma : |\delta(q, a)| \geq 1$. A language L is called *regular* if it is accepted by an NFA.

Finite tree automata. A k-ary finite *tree automaton* (TA) over an alphabet Σ is a quadruple $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ where Q is a finite set of *states*, $\delta : Q^k \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is a set of *leaf states*, and $R \subseteq Q$ is a set of *root* states. We use $(q_1, \ldots, q_k) \xrightarrow{a} s$ to denote that $s \in \delta((q_1, \ldots, q_k), a)$. For a better readability, we often use a notation $\delta_a(q_1, \ldots, q_k)$ instead of $\delta((q_1, \ldots, q_k), a)$. A *run* of $\mathcal{A}$ on a tree τ is a total map $\rho : \mathrm{dom}(\tau) \to Q$ such that if $\tau(p) = \bot$, then $\rho(p) \in I$, else $(\rho(p.\mathtt{1}), \ldots, \rho(p.\mathtt{k})) \xrightarrow{a} \rho(p)$ with $a = \tau(p)$. The run ρ is *accepting* if $\rho(\epsilon) \in R$, and the *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all trees on which $\mathcal{A}$ has an accepting run. $\mathcal{A}$ is a *(bottom-up) deterministic* TA (DTA) if $|I| = 1$ and $\forall q_1, \ldots, q_k \in Q, a \in \Sigma : |\delta((q_1, \ldots, q_k), a)| \leq 1$, and *complete* if $I \geq 1$ and $\forall q_1, \ldots, q_k \in Q, a \in \Sigma : |\delta((q_1, \ldots, q_k), a)| \geq 1$. A language L is called *tree regular* if it is accepted by a TA.

Büchi automata. A (nondeterministic) *Büchi automaton* (BA) over Σ is a quadruple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of *states*, δ is a *transition function* $\delta\colon Q \times \Sigma \to 2^Q$,

and $I, F \subseteq Q$ are the sets of *initial* and *accepting* states, 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 $q \in \delta(p, a)$. Moreover, we extend δ to sets of states $P \subseteq Q$ as $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$. 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 $\inf(\rho)$ denote the states occurring in ρ infinitely often. We say that ρ is *accepting* iff $\inf(\rho) \cap F \neq \emptyset$. 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 drop the superscript when the context is obvious. $\mathcal{A}$ is called *deterministic* if $|I| = 1$ and $\forall q \in Q$ and $\forall a \in \Sigma : |\delta(q, a)| \leq 1$ and $\mathcal{A}$ is called *complete* if for every state q and symbol a, it holds that $|\delta(q, a)| \geq 1$.

For all types of automata, we sometimes fix the alphabet and omit it from the definition. For an NFA/TA/BA $\mathcal{A}$, we also use $|\mathcal{A}|$ to denote the number of states of $\mathcal{A}$.

2.3 Operations on Finite Automata

Finite word automata. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a NFA. Further, let $\cong$ be an equivalence on Q. The *quotient* of $\mathcal{A}$ w.r.t $\cong$ is the automaton $\mathcal{A}/\cong = (Q/\cong, \Sigma, \delta_\cong, I_\cong, F_\cong)$ with the transition function $\delta_\cong([q]_\cong, a) = \{[r]_\cong \mid r \in \delta([q]_\cong, a)\}$ and the set of initial and accepting states $I_\cong = \{[q]_\cong \in Q/\cong \mid q \in I\}$ and $F_\cong = \{[q]_\cong \in Q/\cong \mid q \in F\}$, respectively.

The reverse automaton of $\mathcal{A}$ is defined as $rev(\mathcal{A}) = (Q, \Sigma, \delta^{-1}, F, I)$ where $\delta^{-1}(q, a) = \{p \mid p \xrightarrow{a} q \in \delta\}$. Further, a deterministic automaton of $\mathcal{A}$ created by a *subset construction* is defined as $det(\mathcal{A}) = (2^Q, \Sigma, det(\delta), \{I\}, det(F))$ where $det(\delta)(S, a) = \bigcup_{q \in S} \delta(q, a)$ and $det(F) = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$. For a complete DFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, the *complement* returns $\mathcal{A}^\complement = (Q, \Sigma, \delta, I, Q \setminus F)$. The binary operators $\circ \in \{\cup, \cap\}$ are implemented through a *product construction* as follows: given an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ and another NFA $\mathcal{A}' = (Q', \Sigma, \delta', I', F')$ (for the case of $\cup$ complete ones) the product construction returns the automaton $\mathcal{A} \circ \mathcal{A}' = (Q \times Q', \Sigma, \delta^\times, I \times I', F^\circ)$ where $\delta^\times((q_1, r_1), a) = \delta(q_1, a) \times \delta'(r_1, a)$, and for $(q, r) \in Q \times Q'$, $(q, r) \in F^\cap \Leftrightarrow q \in F \wedge r \in F'$ and $(q, r) \in F^\cup \Leftrightarrow q \in F \vee r \in F'$. The *disjoint union* of $\mathcal{A}$ and $\mathcal{A}'$ is defined as $\mathcal{A} \uplus \mathcal{A}' = (Q \uplus Q', \Sigma, \delta \uplus \delta', I \uplus I', F \uplus F')$.

Finite tree automata. For a complete TA $\mathcal{A} = (Q, \Sigma, \delta, I, R)$, the *complement* assumes that $\mathcal{A}$ is deterministic and returns $\mathcal{A}^\complement = (Q, \Sigma, \delta, I, Q \setminus R)$, and the *subset construction* returns the deterministic and complete automaton $det(\mathcal{A}) = (2^Q, det(\delta), \{I\}, det(R))$ where $det(\delta_a)(S_1, \ldots, S_k) = \bigcup_{q_1 \in S_1, \ldots, q_k \in S_k} \delta_a(q_1, \ldots, q_k)$ and $det(R) = \{S \subseteq Q \mid S \cap R \neq \emptyset\}$. The binary operators $\circ \in \{\cup, \cap\}$ are implemented (as in the case of NFAs) through a *product construction*, which—given a TA $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ and another TA $\mathcal{A}' = (Q', \Sigma, \delta', I', R')$ (for the case of $\cup$ complete ones)—returns the automaton $\mathcal{A} \circ \mathcal{A}' = (Q \times Q', \Sigma, \delta^\times, I \times I', R^\circ)$ where $\delta_a^\times((q_1, r_1), \ldots, (q_k, r_k)) = \delta_a(q_1, \ldots, q_k) \times \delta'_a(r_1, \ldots, r_k)$, and for $(q, r) \in Q \times Q'$, $(q, r) \in R^\cap \Leftrightarrow q \in R \wedge r \in R'$ and $(q, r) \in R^\cup \Leftrightarrow q \in R \vee r \in R'$.

Part I:
Approximate Reduction of Finite Automata for NIDS

Chapter 3

Finite Automata in Network Intrusion Detection

Computer networks came a long way from their beginnings in 1960s. A connection of several computers across the U.S. universities at that time have become an inseparable part of our everyday lives. A trend of these days is an increasing number of communication devices in computer networks (and particularly the Internet). The connected devices are currently not only general-purpose computers but also specialized smart devices such as autonomous cars, thermostats, or emergency notification systems. The later mentioned smart devices are covered by the term *Internet of Things*, which has a great merit on the growing number of connected devices. The Cisco company estimates that by 2023 there will be 29.3 billion networked devices (raising from 18.4 billion in 2018) [2].

A dark side of this development is security. The more networked devices communicate over various protocols, the more opportunities a malicious user has to intrude the systems. The malicious activities vary from e-mail hacking, DoS (denial of service) attacks to cyber espionage and attacks on a critical infrastructure. Consequences of such threats may be disastrous. For example, in 2016 a massive attack against the Ukrainian power grid caused a widespread blackout [19]. In 2020 a cyberattack on computer systems of Universal Health Service in the U.S. caused two days of service outage [81]. The security of computer networks and, consequently, the security of network traffic is hence a crucial task.

The network threats (and their economical and/or political consequences) have led to an enormous effort that has been put into network security in recent years. Defense against the threats includes various techniques with different use-cases ranging from communication encryption through firewall deployment to *(network) intrusion detection systems* (NIDSes). A prominent approach to malicious traffic detection, which is also used within NIDSes, is gathering of high-level information about transmitted packets with a detection based on reasoning about the gathered information. For example, a detection system builds a golden communication profile corresponding to valid traffic and compares this profile with the traffic under inspection [27]. Or, and this is the case that is the most interesting for us, the content of a packet is matched against a series of patterns occurring within the already known attacks [269, 17]. Efficient network pattern matching is usually implemented using finite automata. Therefore, it is necessary to handle the automata in an efficient way. In this chapter, we briefly discuss the use of automata for NIDS, especially for hardware-accelerated pattern matching, including problems that the acceleration involves.

```
/^Attached\s+through\s+port\x3a/smi
/^DmInf\^[^\r\n]*\^\d+\x2E\d+\x2E\d+\x2E\d+\^/smi
/\x5BDRIVE\s+LIST\x5D/smi
```

Figure 3.1: An example of regular expressions in the PCRE format used in SNORT to describe malicious traffic.

Chapter outline. This chapter serves as a brief introduction to an application of finite automata in network intrusion detection systems. Section 3.1 discusses approaches to detection of malicious traffic using finite automata. Section 3.2 deals with hardware-accelerated pattern matching based on finite automata. Finally, Section 3.3 presents state-of-the-art methods of automata reduction.

3.1 Network Intrusion Detection Systems

The blossom of computer networks brought new demands to secure users, data, and infrastructure. At the beginning, network administrators were detecting intrusion or other malicious activities manually by checking the audit logs [212]. Since then, the networks grew up substantially and the malicious activities became much frequent. It is hence impossible to manually detect these activities nowadays, and so it is necessary to have a specialized automatic system tailored for this task. This is the job for a NIDS.

Based on a permission policy, NIDSes are divided into anomaly-based systems and signature-based systems. Anomaly-based systems (sometimes denoted as NIPS) allow only those communications corresponding to normal behavior. Every unusual activity that is not considered as normal is declined. A model of normal communication is stored to the NIDS. Contrary, signature-based systems contain a description of patterns of malicious traffic and every communication that matches a pattern is blocked and an administrator is alerted (see, e.g., [212] for more details). NIDSes involve various techniques and approaches to detect malicious traffic (e.g., genetic algorithms [27], machine learning [252, 160], or signature matching [269, 17, 260]). In this thesis, we will focus on signature-based systems (and simply use NIDSes to refer to those).

Despite the fact that a lot of network traffic is currently encrypted, which narrows down possibilities of *deep packet inspection*, NIDSes are still widely used (e.g., at the entry points of private networks, after the communication is decrypted). SNORT [269], BRO [284], or Suricata [205] are the most popular NIDSes. The patterns describing malicious traffic are usually defined in the form of rules. The rules specify, among others, IP addresses, ports, or contents of packets representing a malicious communication [269, 15].

A prominent way of deep packet inspection is a description of attacks using rules containing *regular expressions* (REs). These REs are then matched against the content (payload) of a packet under inspection. If a match is found, a malicious activity is detected. An example of such REs used in SNORT are shown in Figure 3.1. In order to cover large portion of threats that can occur, a lot of REs are necessary. With an increasing transmission speed of networks, RE matching is a bottleneck of intrusion detection. Currently, high-speed 100 Gbps networks have become more available [28]. On top of that, 200 Gbps and 400 Gbps Ethernet standards were recently approved [1]. In order to examine traffic in 100 Gbps networks in software, it is necessary to build a special distributed NIDS cluster [278]. Existing single-box software NIDSes alone are not capable to handle 100 Gbps (and faster) traffic. Their throughput ranges from below 1 Gbps [38] up to 33 Gbps employ-

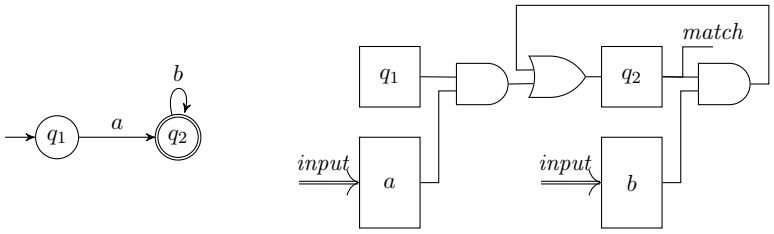


Figure 3.2: An NFA accepting the language ab^* and its corresponding high-level hardware implementation [254]. The states are realized by flip-flops q_1 and q_2. An input symbol is selected in a decoder (output is high if input corresponds to a or b, respectively). The transition function is then represented by and/or logical gates.

ing a combination of hi-tech multicore CPUs and manycore GPUs [156]. To tackle packet inspection in high-speed networks, hardware-accelerated solutions were introduced to speed up RE matching of packet payloads [254, 204, 177].

3.2 Hardware-Accelerated Pattern Matching

As we outlined in the previous section, a specialized hardware solution performing RE matching is necessary for high-speed networks. The idea of hardware acceleration is to offload the difficult task of pattern matching from SW to HW. In this section, we briefly discuss approaches for HW-accelerated pattern matching of network traffic.

From a high-level point of view, the REs describing suspicious traffic are first translated into a deterministic/nondeterministic automaton $\mathcal{A}$. The HW unit then employs $\mathcal{A}$ for pattern matching. In particular, the pattern matching unit checks, an incoming packet if its payload belongs to $\mathcal{L}(\mathcal{A})$. If a match is found, the suspicious packet may be send for further inspection to software (the HW unit can be hence seen as a traffic pre-filter). According to the background automaton model we recognize *DFA-based* and *NFA-based* acceleration.

The matching units are often based on the *Field Programmable Gate Array* (FPGA) technology (especially for the NFA-based acceleration) [254, 79, 204]. The FPGA technology allows users to specify their own behavior of a logical circuit. The functionality is described using specialized languages (e.g., VHDL or Verilog). A logical circuit is then synthesized into an FPGA unit according to the description. An FPGA unit may be reprogrammed multiple times. Note that although FPGAs are widely used, it is not the only possible platform. Pattern matching acceleration approaches aim also, e.g., on GPUs [72, 287].

DFA-based acceleration. DFA-based acceleration of RE matching employs deterministic finite automata representing the regular expressions. The used deterministic model allows to check an input string w for a match in time $\mathcal{O}(|w|)$. The transition function of the DFA is stored in a memory, which allows to dynamically and quickly update/change the used DFA. On the other hand, the HW unit realizing pattern matching of the input string w requires a memory with a low latency. Access to the memory is not the only issue. Another issue is that a DFA can be exponentially larger than its nondeterministic counterpart. DFA-based acceleration is hence memory intensive. A lot of effort has been put into reduction of this memory bottleneck. The amount of used memory can be reduced by adjusting the deterministic automaton model allowing higher rate of compression, such as optimized

DFAs with default transitions [180, 181], δDFAs with efficient transition storage [110], hybrid DFAs [35], automata with a scratch memory [257], automata with counting constraints [37], or multi-stride automata [36]. Other approaches aiming at this issue include leveraging properties of common REs [179], rewriting and grouping of REs [299] and/or uses specialized architectures with perfect hashing [164, 165, 166], pipelined automata [62], or efficient architectures consuming more symbols per clock cycle [295].

NFA-based acceleration. DFA-based acceleration suffers from state explosion during the determinization of NFAs corresponding to REs as well as from memory access latency. These issues limit DFA-based approach from use in high-speed networks with a huge set of REs. A straightforward generalization of the DFA matching unit to NFAs with transition function stored in the memory does not solve the problems, because the complexity of a string matching would be $\mathcal{O}(n|w|)$ where n is the number of NFA states. The NFA-based acceleration alleviates these issues by a direct synthesis of the transition function into a reconfigurable HW unit (usually an FPGA unit). Such a construction, introduced in [254], allows to check an input string for a match in time $\mathcal{O}(|w|)$. The states of the NFA are represented by flip-flops and the transitions between them are composed by logical gates. An example of an NFA implementation is shown in Figure 3.2. A disadvantage of the direct synthesis is impossibility of a quick change of the used REs and also considerable restrictions of FPGA resources. The construction of [254] was further optimized by a shared character decoder [79], RE subexpressions sharing [261, 192] or an efficient state encoding [301]. These methods alone are, however, insufficient for high-speed networks (100 Gbps and beyond). In order to achieve a better throughput, various techniques have been introduced, including multicharacter matching using spatial stacking [296] or multi-stride architecture [36]. However, as shown in [204], the multi-striding does not scale well and hence it is not sufficient for 100 Gbps throughputs. To achieve 100 Gbps throughputs, an FPGA-based architecture with pipelined automata was proposed in the work [204]. Nevertheless, such a speed requires massive parallelization, which reduces the FPGA resources. It is, therefore, a matter of uttermost importance to keep the NFA obtained from the REs as small as possible.

For both DFA-based and NFA-based acceleration, it is apparent that the smaller the automata we obtain from the REs, the higher throughput we can achieve (due to a possibility of parallelization). And, consequently, we use less memory and/or less FPGA resources. Efficient automata reduction is hence important particularly for high-speed HW-accelerated pattern matching. In the following section, we, therefore, discuss techniques for reduction of DFAs and NFAs.

3.3 Reduction of Finite Automata

Reduction of automata size is a crucial task for many applications dealing with this computational model, not only for monitoring of network traffic, where automata are used to represent attacks and protocols but also for verification of parametric and infinite-state systems [55, 54], software regular expression matching [271, 262], or in various automata decision procedures [65, 143, 100]. When talking about a reduction of finite automata, we can distinguish between a *transition*-based reduction and a *state*-based reduction. In this thesis, we are interested in the state-based reduction, i.e., reduction of the number of states. In this section, we discuss basic approaches for reduction of finite automata.

3.3.1 Reduction of DFAs

First, we focus on reduction of deterministic automata. In contrast to general NFAs, it is a practically feasible operation to find an equivalent DFA with a minimum number of states for a given DFA. The existence of such a minimal (and, on top of that, canonical) automaton follows from Myhill-Nerode theorem. A DFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is the minimal DFA representing the regular language $\mathcal{L}(\mathcal{A})$ if no two states $p, q \in Q$ are *indistinguishable*, in other words, there are no two states that "behave the same" on each word. Note that we assume that the automaton is *trimmed*, i.e., all states are reachable from an initial state and all states can reach a final state. Formally, states $p, q \in Q$ are *indistinguishable*, denoted as $p \equiv_{\mathsf{u}} q$, iff for each $x \in \Sigma^*$ and $p \overset{x}{\rightsquigarrow} p'$, $q \overset{x}{\rightsquigarrow} q'$ we have $p' \in F \iff q' \in F$ [150]. The indistinguishability relation is an equivalence on Q and, moreover, quotienting of $\mathcal{A}$ w.r.t. $\equiv_{\mathsf{u}}$ preserves the language of $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\equiv_{\mathsf{u}})$. Therefore, for a DFA $\mathcal{A}$, the minimal equivalent DFA can be constructed as $\mathcal{A}/\equiv_{\mathsf{u}}$.

It remains to discuss a computation of the indistinguishability relation to make the picture of minimization complete. Hopcroft's algorithm and Moore's algorithm are two widely used algorithms to compute the indistinguishability relation [63, 216]. Both these algorithms work by an iterative refinement of an approximation of $\equiv_{\mathsf{u}}$. The worst-case time complexity of Hopcroft's algorithm is $\mathcal{O}(n \log n)$. The worst-case time complexity of Moore's algorithm, originally presented for Moore's automata, is $\mathcal{O}(n^2)$. The average time complexity of both algorithms is $\mathcal{O}(n \log \log n)$ [92] where n is the number of states of an input DFA.

A slightly different approach to DFA minimization (than computing the indistinguishability relation) was proposed by Brzozowski [63]. Brzozowski's algorithm uses properties of reverse automata. In particular, for a DFA $\mathcal{A}$ accepting the language L, the trimmed automaton $det(rev(\mathcal{A}))$ is the minimal DFA for the reverse of language L[1]. If we then apply *det* and *rev* once again we obtain the minimal DFA for L. Note that this approach works as well if $\mathcal{A}$ is nondeterministic. Although the worst-case time complexity is exponential, it is efficient in some cases [43]. Except the general aforementioned approaches, there are minimization techniques for special cases of deterministic automata [43, 243, 34].

3.3.2 Reduction of NFAs

In the second part, we focus on the reduction of general nondeterministic finite automata, where the situation is more involved. It is, of course, possible to first determinize and minimize the input NFA. The determinization may, however, cause an exponential blowup, which is limiting for many huge NFAs used in practice. Moreover, even this minimal DFA can be in the worst case exponentially larger than a minimal NFA (and hence also exponentially larger than the input NFA). On the other side, minimization of NFAs is a **PSPACE**-complete problem with, up to our knowledge, no known good heuristics, making it infeasible to apply in real-world applications [159].

Nevertheless, there are feasible approaches for NFA reduction based on the concept of (bi)simulation. These reductions use structural properties of NFAs, so they cannot guarantee obtaining a minimal NFA, but in many practical situations significantly reduce the input automata. Intuitively, (bi)simulations capture the structural similarity between various parts of automata. In the following text, we fix an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ having n states and m transitions. Formally, for the NFA $\mathcal{A}$ a *bisimulation* is a relation $\equiv_{\mathsf{b}} \subseteq Q \times Q$

[1]The reverse of language of L contains reversed words from L.

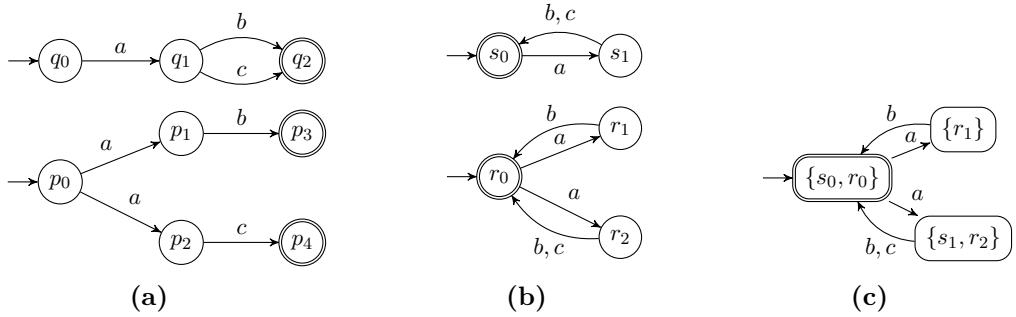


Figure 3.3: An example of two NFAs over $\{a, b, c\}$ and a reduced NFA with merged states.

s.t. $p \equiv_{\mathsf{b}} q$ iff (i) $p \in F \iff q \in F$ (ii) $\forall a \in \Sigma : \; q \xrightarrow{a} q' \implies \exists p' \in Q : p \xrightarrow{a} p' \wedge q' \equiv_{\mathsf{b}} p'$ (iii) $\forall a \in \Sigma : \; p \xrightarrow{a} p' \implies \exists q' \in Q : q \xrightarrow{a} q' \wedge q' \equiv_{\mathsf{b}} p'$ [25]. The bisimulation is an equivalence relation implying language equivalence of states and, moreover, the quotient of $\mathcal{A}$ w.r.t. $\equiv_{\mathsf{b}}$ preserves the language, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\equiv_{\mathsf{b}})$. For deterministic automata the maximum bisimulation coincides with the indistinguishability relation. Bisimulation is also a sound technique for language equivalence checking of two NFAs (completeness requires at least a partial determinization [149, 52]). Bisimulation can be computed using an algorithm of iteratively refining partitions until the fixpoint is reached. The partition at the fixpoint corresponds to the bisimulation. The time complexity of the algorithm is $\mathcal{O}(m \log n)$ [25].

Although bisimulation equivalence is a cheap technique for reduction of NFAs, the relation is, however, often quite coarse and, therefore, the reduction is small. In order to obtain a greater reduction, we can use a simulation relation, which is weaker than bisimulation. For an NFA $\mathcal{A}$ a *forward simulation* is a relation $\sqsubseteq_{\mathsf{fw}} \subseteq Q \times Q$ s.t. $p \sqsubseteq_{\mathsf{fw}} q$ if (i) $p \in F \implies q \in F$ and (ii) $\forall a \in \Sigma : p \xrightarrow{a} p' \implies \exists q' \in Q : q \xrightarrow{a} q' \wedge p' \sqsubseteq_{\mathsf{fw}} q'$. Intuitively, forward simulation between two states $p \sqsubseteq_{\mathsf{fw}} q$ expresses that for each word w a (forward) run from p over w can be mimicked by a run from q over w. For an NFA there is a unique maximal forward simulation called the *forward simulation preorder* denoted as $\preccurlyeq_{\mathsf{fw}}$. Forward simulation between two states implies inclusion of their languages (but not vice versa as shown in Example 3.3.1) and hence it can be used in optimizations of inclusion checking of two NFAs [14].

Example 3.3.1. *Consider the NFA in Figure 3.3a. In this automaton we have* $\mathcal{L}(q_0) = \mathcal{L}(p_0)$*, however,* $q_0 \not\preccurlyeq_{\mathsf{fw}} p_0$*. This is because at the time of choosing a transition over a from* q_0 *we do not know if the following transition will be over b or c. Therefore, we cannot ensure a correct choice of a transition from* p_0*.*

A dual notion to forward simulation is a backward simulation. For an NFA $\mathcal{A}$ a *backward simulation*, denoted as $\sqsubseteq_{\mathsf{bw}}$, is a forward simulation on $rev(\mathcal{A})$. A backward simulation between two states implies inclusion of their backward languages. For an NFA there is a unique maximal backward simulation called the *backward simulation preorder* denoted as $\preccurlyeq_{\mathsf{bw}}$. The *forward/backward simulation equivalence* for a forward/backward preorder $\preccurlyeq$ is then the maximal symmetric fragment of $\preccurlyeq$, i.e., it is given as $\preccurlyeq \cap \preccurlyeq^{-1}$. The quotient of $\mathcal{A}$ w.r.t. a simulation equivalence does not affect the language of $\mathcal{A}$ and hence it can be used to reduce the automata. Moreover, the simulation reduction provides greater reduction compared to the bisimulation reduction since two simulation equivalent states need not be bisimulation equivalent as shown in Example 3.3.2.

Example 3.3.2. *Consider an NFA in Figure 3.3b. States s_0 and r_0 forward-simulate each other in forward manner, i.e., $s_0 \preccurlyeq_{\mathsf{fw}} r_0$ and $r_0 \preccurlyeq_{\mathsf{fw}} s_0$. However, since $r_1 \not\equiv_{\mathsf{b}} s_1$, it also holds that $r_0 \not\equiv_{\mathsf{b}} s_0$. The quotient of this automaton w.r.t. $\equiv_{\mathsf{b}}$ does not bring any reduction. The quotient w.r.t. the forward simulation equivalence merges state s_0 with r_0 and state s_1 with r_2 yielding the NFA in Figure 3.3c having 3 states only.*

There are several approaches to compute the simulation preorder, based on a stepwise refinement of the preorder approximation [144], or on a refinement of the complementary simulation relation [153]. The time complexity of these algorithms is $\mathcal{O}(mn)$. Moreover there are algorithms improving upon [144] with slightly better time and space complexity [88, 238].

The simulation reduction described above can be further tuned in order to obtain better reduction results. Merging of states can be steered not by the (bi)simulation equivalence, but directly by the simulation preorders [153, 154]. In particular, two states p, q can be merged if any of these conditions is met: (i) $p \preccurlyeq_{\mathsf{fw}} q$ and $q \preccurlyeq_{\mathsf{fw}} p$ (ii) $p \preccurlyeq_{\mathsf{fw}} q$ and $q \preccurlyeq_{\mathsf{bw}} p$ (iii) $p \preccurlyeq_{\mathsf{fw}} q$, $p \preccurlyeq_{\mathsf{bw}} q$, and $\neg\exists w \in \Sigma^+ : p \overset{w}{\rightsquigarrow} p$ [154]. Unlike the equivalence-based reduction, in this case the preorders $\preccurlyeq_{\mathsf{fw}}$ and $\preccurlyeq_{\mathsf{bw}}$ must be updated after each merge to reflect the changes. Another approach, combining the simulation preorders into a mediated preorder was studied in [5, 4]. The mediated preorder M is the maximal preorder s.t. $\preccurlyeq_{\mathsf{bw}} \subseteq M \subseteq \preccurlyeq_{\mathsf{fw}}^{-1} \circ \preccurlyeq_{\mathsf{bw}}$. An equivalent reduced NFA can be then constructed by quotient of $\mathcal{A}$ w.r.t. $M \cap M^{-1}$.

Simulation relation can be further extended with an additional information about future moves of a simulation smaller state. The bigger state sees some bounded future and it may hence adjust a choice of a transition accordingly (or possibly choose more transitions). Simulation relation can then be, according to the type of information, generalized to step simulation, lookahead simulation, or multipebble simulation [206, 108]. Yet, in order to achieve better reduction results, (generalized) simulation-based reduction can be further combined with techniques for adding or removing transitions [206, 68].

3.3.3 Language Non-preserving Reduction

So far, we were dealing only with language preserving reductions, i.e., the reduced NFA has the same language as the original one. For some applications even a minimal NFA may, however, be too large (e.g., when the number of states is limited by hardware resources) or the automata themselves approximate languages that need not be regular at all (e.g., in verification of infinite state systems). In the context of such applications, it makes sense to talk about language non-preserving (approximate) reductions.

A problem of an approximation of finite languages was studied in [69, 176, 225]. A finite language L is approximated by a DFA $\mathcal{A}$ s.t. $\mathcal{L}(\mathcal{A}) \cap \Sigma^{\leq \ell} = L$ where ℓ is the length of the longest string from L. A DFA $\mathcal{A}$ satisfying this criterion is called the *cover automaton* of L. The target is to obtain a minimal cover automaton. This problem can be efficiently solved in polynomial time [176].

Another problem involving approximate reduction is called *hyperminimization* [122, 198, 24]. For a given DFA $\mathcal{A}$, the aim of hyperminimization is to find a minimal DFA $\mathcal{A}'$ s.t. $\mathcal{L}(\mathcal{A}') \triangle \mathcal{L}(\mathcal{A})$ is a finite set. Since $\mathcal{L}(\mathcal{A}')$ can differ from $\mathcal{L}(\mathcal{A})$ on a finite number of strings, there is a space for a more aggressive reduction, which can lead to smaller automata. In particular, there is an efficient polynomial algorithm constructing a minimal $\mathcal{A}'$ satisfying the symmetric difference constraint [122].

Approximation of deterministic automata in a slightly different setting was studied in [120]. The task is, for a given DFA $\mathcal{A}$, to find a DFA $\mathcal{A}'$ with at most k states approximating $\mathcal{A}$. The approximated DFA $\mathcal{A}'$ is constructed by collapsing states of $\mathcal{A}$. The quality of the approximation is measured by (i) the number of misclassified prefixes (up to some length), or (ii) the sum of probabilities of misclassified words (w.r.t. Exponential distribution). The automata approximation w.r.t. an error measure is then formulated as an optimization problem solved with proposed incomplete greedy approaches.

Apart from the aforementioned general approaches for approximate (language non-preserving) reductions, there appeared techniques involving approximate reduction tailored for a concrete application. In [194] an approximation of DFAs is used to create a tree of approximated DFAs accelerating software-based regular expression pattern matching of attacks in a network traffic. Finally, in the context of software verification, approximation of automata representing configurations of a system is used as an abstraction technique for regular model checking [55, 54].

Chapter 4

Approximate Reduction of NFAs with Formal Guarantees

Recall from Chapter 3 that RE matching is the most computationally demanding task of traffic inspection as its cost grows with the speed of the network traffic as well as with the number and complexity of the REs being matched. The current single-box software-based NIDSes cannot perform the RE matching on high-speed networks beyond 100 Gbps. We also mentioned that a promising approach to speed up NIDSes is to offload RE matching into hardware. The hardware then serves as a pre-filter of the network traffic, discarding the majority of the packets from further processing.

Since DFA-based acceleration (see Section 3.2) suffers from the state space explosion during the determinization (causing subsequent memory issues), we focus at NFA-based acceleration using FPGAs. Due to their inherent parallelism, FPGAs provide an efficient way of implementing NFAs, which naturally arise from the input REs. 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 FPGA (we assume the architecture of [204]); each of them implementing a separate copy of the concerned NFA, and so reducing the size of the NFAs is of utmost importance. Various language-preserving automata reduction approaches exist, mainly based on computing (bi)simulation relations on automata states (cf. Section 3.3.2). The reductions they offer, however, do not satisfy the needs of high-speed hardware-accelerated NIDSes.

Our answer to the problem, that we propose in this chapter, is *approximate reduction of NFAs with formal guarantees*, allowing for a trade-off between the achieved reduction and the precision of the RE matching. To formalize the intuitive notion of precision, we propose a novel concept of *probabilistic distance* of automata. Probabilistic distance 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. Based on the notion of precision, we propose language-nonpreserving reductions (i) minimizing the NFA size w.r.t. the given maximum distance from the original NFA, or (ii) minimizing the distance w.r.t. the given maximum number of NFA states. Since the exact proposed reductions are computationally demanding, we propose greedy algorithms utilizing the usual structure of NFAs obtained from REs.

Overview of the proposed approach. In the first part we deal with the probabilistic distance of two NFAs measuring error (precision) of approximate reduction. The probabilistic distance is expressed as the probability of words that belong to the symmetric difference of the languages of given NFAs.

Having formalized the notion of precision, we specify, as already mentioned, the target of our reductions as two variants of an optimization problem: (i) minimizing the NFA size given the maximum allowed error (distance from the original), or (ii) 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 REs 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 precomputed *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 hardware NFAs 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 REs 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 FPGA chip real-life REs encoding malicious traffic, allowing them to be used with a negligible error for filtering at speeds of 100 Gbps (and even 400 Gbps).

Related work. Hardware acceleration for RE matching at the line rate is an intensively studied technology that uses general-purpose hardware as well as FPGAs (see Section 3.2). A lot of works focus on DFA implementation and optimization techniques (see Section 3.2, paragraph related to DFA-based acceleration). 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. Therefore, FPGAs combined with NFA-based acceleration is suitable for high-speed networks (see Section 3.2 the part related to NFA-based acceleration).

In [194], which is probably the work closest to ours, the authors consider a set of REs 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 (i) the approach targets approximation of DFAs (not NFAs), (ii) the over-approximation is based on a given traffic sample only (it cannot benefit from a probabilistic model), and (iii) no probabilistic guarantees on the approximation error are provided.

Language-nonpreserving reduction of DFAs, in particular hyperminimization and the DFA approximation of [120], was discussed in Section 3.3.3. Neither of these approaches, however, considers reduction of NFAs nor allows to control the expected error with respect to the real traffic.

In addition to the metrics discussed in the context of language-nonpreserving reduction in Section 3.3.3, the following metrics should also be mentioned. The Cesaro-Jaccard distance studied in [224] is, in spirit, similar to [120] and also does not reflect the probability of individual words. The edit distance of weighted automata from [213] 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 [18], 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.

Although, language-preserving minimization of a given NFA is a **PSPACE**-complete problem, there are more feasible (polynomial-time) size reductions of NFAs based on (bi)simulations (but not only), which do not aim for a truly minimal NFA (see Section 3.3.2 for more details). 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 REs 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.

Chapter outline. This chapter is structured as follows. Section 4.1 contains necessary definitions used in the rest of the chapter. Section 4.2 deals with probabilistic distance of two NFAs. Section 4.3 formalizes the proposed approximate reductions and Section 4.4 focuses on a heuristic approach to approximate reduction. Finally, Section 4.5 contains experimental evaluation and Section 4.6 concludes the chapter.

4.1 Probabilistic Automata

In this section, we give definitions used in the rest of the chapter. We assume the definitions presented in Chapter 2 and we extend them to fit our needs.

Sets, vectors, and matrices. We use $\langle a, b\rangle$ to denote the set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$. We use $\boldsymbol{A}$ to denote a matrix, and $\boldsymbol{A}^\top$ for its transpose, and $\boldsymbol{I}$ for the identity matrix. We use the notation $[v_1, \ldots, v_n]$ to denote a row vector of n elements, $\mathbf{1}$ to denote the all 1's (column) vector $[1, \ldots, 1]^\top$ (the dimension of $\mathbf{1}$ is always clear from the context).

Finite automata. In this chapter we fix an alphabet Σ. Let $\mathcal{A} = (Q, \delta, I, F)$ be an NFA over Σ. $\mathcal{A}$ is called *unambiguous* (UFA) if $\forall w \in \mathcal{L}(\mathcal{A}) : \exists! q_I \in I, \rho \in Q^*, q_F \in F : q_I \overset{w,\rho}{\rightsquigarrow} q_F$. 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 $\mathit{trim}(\mathcal{A}) = \mathcal{A}_{|C}$ 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 $\mathit{reach}(R)$ to denote the set of states reachable from R, $\mathit{reach}(R) = \{r' \mid \exists r \in R : r \rightsquigarrow r'\}$.

Probabilistic automata. 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 $\mathit{supp}(\mathcal{P}) = (Q[\mathcal{P}], \delta[\mathcal{P}], I[\mathcal{P}], F[\mathcal{P}])$ s.t.

$$\delta[\mathcal{P}] = \{(i, a, j) \mid \boldsymbol{\Delta}_a[i, j] > 0\}, \quad I[\mathcal{P}] = \{i \mid \boldsymbol{\alpha}[i] > 0\}, \quad F[\mathcal{P}] = \{i \mid \boldsymbol{\gamma}[i] > 0\}.$$

Let us assume that every PA $\mathcal{P}$ is such that $\mathit{supp}(\mathcal{P}) = \mathit{trim}(\mathit{supp}(\mathcal{P}))$. 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 $\boldsymbol{\Delta}_\epsilon = \boldsymbol{I}$. 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 $\mathcal{P}_{Exp} = \left(\mathbf{1}, [\mu], \{[\mu]_a\}_{a \in \Sigma}\right)$ 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 $\mathcal{P}_{Exp}$ in the proofs is to assign every word $w \in \Sigma^*$ the (non-zero) probability $\Pr_{\mathcal{P}_{Exp}}(w) = \mu^{|w|+1}$; any other PA assigning non-zero 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 [253].

4.2 Probabilistic Distance

In this section, we first introduce the key notion of our approach: a *probabilistic distance* of a pair of finite automata w.r.t. 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*.

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 $\mathcal{A}_2$ w.r.t. $\Pr_{\mathcal{P}}$ is defined as

$$d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2) = \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1) \triangle \mathcal{L}(\mathcal{A}_2)).$$

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* [94].)

The value of $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1) \triangle \mathcal{L}(\mathcal{A}_2))$ can be computed as follows. Using the fact that (i) $L_1 \triangle L_2 = (L_1 \setminus L_2) \uplus (L_2 \setminus L_1)$ and (ii) $L_1 \setminus L_2 = L_1 \setminus (L_1 \cap L_2)$, we get

$$\begin{aligned} d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2) &= \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}_2)) + \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_2) \setminus \mathcal{L}(\mathcal{A}_1)) \\ &= \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1) \setminus (\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2))) + \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_2) \setminus (\mathcal{L}(\mathcal{A}_2) \cap \mathcal{L}(\mathcal{A}_1))) \\ &= \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1)) + \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_2)) - 2 \cdot \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)). \end{aligned}$$

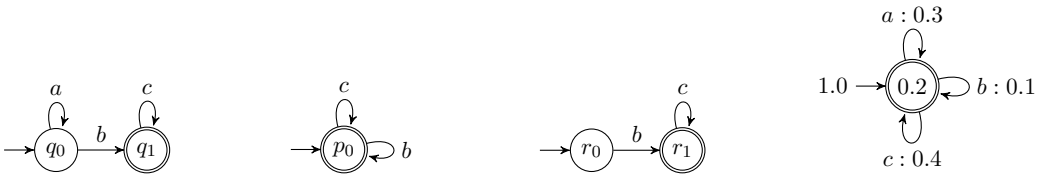
Hence, the key step is to compute $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ for an NFA $\mathcal{A}$ and a PA $\mathcal{P}$. Problems similar to computing such a probability have been extensively studied in several contexts including verification of probabilistic systems [280, 26, 39].

In our approach, we apply the method of [39] and compute $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ in the following way. We first check whether the NFA $\mathcal{A}$ is unambiguous. This can be done by using the product construction 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. $p \neq q$ [214]. If $\mathcal{A}$ is ambiguous, we either determinize it or disambiguate it [214], 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 $\mathcal{L}(\mathcal{A})$ but also their probability in $\mathcal{P}$ (we give the formal definition of $\mathcal{R}$ inside the proof of Lemma 4.2.2). 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 4.2.1) 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 $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$, 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 $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ despite $\mathcal{R}$ being a PPA. The following example shows a computation of the probabilistic distance.

Example 4.2.1. *Consider NFAs and a PA over $\Sigma = \{a, b, c\}$ given in Figure 4.1. To compute the value of $d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2)$ we first need to compute the values $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1))$, $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_2))$,*

[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.



(a) NFA $\mathcal{A}_1$ **(b)** NFA $\mathcal{A}_2$ **(c)** NFA $\mathcal{A}_1 \cap \mathcal{A}_2$ **(d)** PA $\mathcal{P}$

Figure 4.1: An example of NFAs and a PA over $\{a, b, c\}$. Transition probabilities of $\mathcal{P}$ are of the form *symbol:probability*. The number in a state denotes the accepting probability.

and $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2))$*. Since automata* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *are both deterministic, we can use the procedure described in the previous paragraph.*

We first compute the value $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1))$*. The product of* $\mathcal{A}_1$ *and* $\mathcal{P}$ *leads to a PPA* $\mathcal{R}_1 = (\boldsymbol{\alpha}_1, \boldsymbol{\gamma}_1, \{\boldsymbol{\Delta}_a^1\}_{a\in\Sigma})$ *where*

$$\boldsymbol{\Delta}_1 = \sum_{a\in\Sigma} \boldsymbol{\Delta}_a^1 = \begin{bmatrix} 0.3 & 0.1 \\ 0 & 0.4 \end{bmatrix}, \qquad \boldsymbol{\alpha}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \boldsymbol{\gamma}_1 = \begin{bmatrix} 0 \\ 0.2 \end{bmatrix}.$$

Then the probability of $\mathcal{L}(\mathcal{A}_1)$ *can be computed as*

$$\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1)) = \boldsymbol{\alpha}_1^\top \cdot (\boldsymbol{I} - \boldsymbol{\Delta}_1)^{-1} \cdot \boldsymbol{\gamma}_1 = [1, 0] \cdot \begin{bmatrix} 0.7 & -0.1 \\ 0 & 0.6 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ 0.2 \end{bmatrix} = \frac{1}{21}.$$

In a similar way we can compute $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_2)) = \frac{2}{3}$ *and* $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2)) = \frac{1}{30}$*. Based on these values the probabilistic distance of* A_1 *and* $\mathcal{A}_2$ *w.r.t.* $\mathcal{P}$ *is given as* $d_{\mathcal{P}}(\mathcal{A}_1, \mathcal{A}_2) = \frac{68}{105} \approx 0.647$*.*

The two lemmas below summarize the complexity of the computation of $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ for NFAs and UFAs, respectively.

Lemma 4.2.1. *Let* $\mathcal{P}$ *be a PA and* $\mathcal{A}$ *an NFA. The problem of computing* $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ *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 $\mathcal{A}'$ is a deterministic automaton obtained from $\mathcal{A}$, i.e., $\mathcal{A}' = det(\mathcal{A})$. The system of equations $\mathcal{S}(\mathcal{A}, \mathcal{P})$ is defined as follows (cf. [39]):

$$\xi_{[R,p]} = \begin{cases} 0 & \text{if } \mathcal{L}_\mathcal{A}(R) \cap \mathcal{L}_{\mathcal{P}'}(p) = \emptyset, \\ \sum\limits_{a\in\Sigma} \sum\limits_{p'\in Q[\mathcal{P}]} (\boldsymbol{\Delta}_a[p, p'] \cdot \xi_{[\delta_\mathcal{A}(R,a),p']}) + \boldsymbol{\gamma}[p] & \text{if } R \cap F_\mathcal{A} \neq \emptyset, \\ \sum\limits_{a\in\Sigma} \sum\limits_{p'\in Q[\mathcal{P}]} \boldsymbol{\Delta}_a[p, p'] \cdot \xi_{[\delta_\mathcal{A}(R,a),p']} & \text{otherwise,} \end{cases}$$

such that $\mathcal{P}' = \mathit{supp}(\mathcal{P})$ and $\delta_{\mathcal{A}}(R, a) = \bigcup_{r \in R} \delta(r, a)$. The test $\mathcal{L}_{\mathcal{A}}(R) \cap \mathcal{L}_{\mathcal{P}'}(p) = \emptyset$ can be performed by checking $\exists r \in R : \mathcal{L}_{\mathcal{A}}(r) \cap \mathcal{L}_{\mathcal{P}'}(p) = \emptyset$, which can be done in polynomial time.

It holds that $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A})) = \sum_{p \in Q[\mathcal{P}]} \boldsymbol{\alpha}[p] \cdot \xi_{[I_{\mathcal{A}}, p]}$. 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 [87, Corollary 2] (i.e., it is in the class **NC**).[3] According to [114, 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 [223, 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.

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 Section 4.1, which assigns every word a non-zero probability. it holds that

$$\mathcal{L}(\mathcal{A}) = \Sigma^* \quad \text{iff} \quad \Pr_{\mathcal{P}_{Exp}}(\mathcal{L}(\mathcal{A})) = 1. \qquad \square$$

Lemma 4.2.2. *Let $\mathcal{P}$ be a PA and $\mathcal{A}$ a UFA. The problem of computing $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ is in **PTIME**.*

Proof. We modify the proof from [39] 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]

$$\begin{aligned}\boldsymbol{\alpha}_{\mathcal{R}}[(q_{\mathcal{P}}, q_{\mathcal{A}})] &= \boldsymbol{\alpha}_{\mathcal{R}}[q_{\mathcal{P}}] \cdot |\{q_{\mathcal{A}}\} \cap I|, \\ \boldsymbol{\gamma}_{\mathcal{R}}[(q_{\mathcal{P}}, q_{\mathcal{A}})] &= \boldsymbol{\gamma}_{\mathcal{R}}[q_{\mathcal{P}}] \cdot |\{q_{\mathcal{A}}\} \cap F|, \\ \boldsymbol{\Delta}^{\mathcal{R}}_a[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] &= \boldsymbol{\Delta}_a[q_{\mathcal{P}}, q'_{\mathcal{P}}] \cdot |\{q'_{\mathcal{A}}\} \cap \delta(q_{\mathcal{A}}, a)|.\end{aligned}$$

[3] We use $\log k$ to denote the base-2 logarithm of k.

[4] We assume an implicit bijection between states of the product $\mathcal{R}$ and $\{1, \ldots, |Q[\mathcal{R}]|\}$.

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 $w \notin \mathcal{L}(A)$, then $\Pr_{\mathcal{R}}(w) = 0$ and (ii) if $w \in \mathcal{L}(\mathcal{A})$ 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 $\mathcal{L}(\mathcal{A})$, i.e., $\Pr_{\mathcal{R}}(w) = \Pr_{\mathcal{P}}(w)$ for all $w \in \mathcal{L}(\mathcal{A})$, so $\mathcal{R}$ can be seen as the restriction of $\Pr_{\mathcal{P}}$ to $\mathcal{L}(\mathcal{A})$. In the following, we assume $\mathcal{R}$ is trimmed.

In order to compute $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$, we construct a matrix $\boldsymbol{E}$ defined as $\boldsymbol{E} = \sum_{a \in \Sigma} \boldsymbol{\Delta}_a^{\mathcal{R}}$. 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, e.g., in [39]). 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 $\boldsymbol{E}^* = \sum_{i=0}^{\infty} \boldsymbol{E}^i = (\boldsymbol{I} - \boldsymbol{E})^{-1}$ [146]. Moreover, note that matrix inversion can be done in polynomial time [258].

$\boldsymbol{E}^*$ represents the reachability between nodes of $\mathcal{R}$, i.e., $\boldsymbol{E}^*[r, r']$ is the sum of significances of all (possibly infinitely 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 $\mathcal{P}$ w.r.t. $\mathcal{L}(\mathcal{A})$, i.e.,

$$\boldsymbol{E}^*[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] = \sum \left\{ \boldsymbol{\Delta}_w[q_{\mathcal{P}}, q'_{\mathcal{P}}] \;\middle|\; q_{\mathcal{A}} \overset{w}{\rightsquigarrow} q'_{\mathcal{A}}, w \in \Sigma^* \right\}. \tag{4.1}$$

We prove Equation (4.1) using the following reasoning. First, we show that

$$\boldsymbol{E}^n[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] = \sum \left\{ \boldsymbol{\Delta}_w[q_{\mathcal{P}}, q'_{\mathcal{P}}] \;\middle|\; q_{\mathcal{A}} \overset{w}{\rightsquigarrow} q'_{\mathcal{A}}, w \in \Sigma^n \right\}, \tag{4.2}$$

i.e., $\boldsymbol{E}^n$ represents the reachability in $\mathcal{P}$ w.r.t. $\mathcal{L}(\mathcal{A})$ for words of length n. We prove Equation (4.2) 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 (4.2) holds for $n > 1$; we show that it holds also for $n + 1$. We start with the following reasoning:

$$\begin{aligned}
\boldsymbol{E}^{n+1}[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] &= (\boldsymbol{E}^n \boldsymbol{E})[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] \\
&= \sum \left\{ \boldsymbol{E}^n[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q''_{\mathcal{P}}, q''_{\mathcal{A}})] \cdot \boldsymbol{E}[(q''_{\mathcal{P}}, q''_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] \;\middle|\; (q''_{\mathcal{P}}, q''_{\mathcal{A}}) \in Q[\mathcal{R}] \right\}.
\end{aligned}$$

The last line is obtained via the definition of matrix multiplication. Further, using the induction hypothesis, we get

$$
\begin{aligned}
&\boldsymbol{E}^{n+1}[(q_{\mathcal{P}}, q_{\mathcal{A}}), (q'_{\mathcal{P}}, q'_{\mathcal{A}})] \\
&\quad = \sum \Bigg\{ \sum \Big\{ \boldsymbol{\Delta}_w[q_{\mathcal{P}}, q''_{\mathcal{P}}] \;\Big|\; q_{\mathcal{A}} \overset{w}{\rightsquigarrow} q''_{\mathcal{A}}, w \in \Sigma^n \Big\} \cdot \sum \Big\{ \boldsymbol{\Delta}_a[q''_{\mathcal{P}}, q'_{\mathcal{P}}] \;\Big|\; q''_{\mathcal{A}} \overset{a}{\to} q'_{\mathcal{A}}, a \in \Sigma \Big\} \;\Bigg|\; \\
&\qquad\qquad (q''_{\mathcal{P}}, q''_{\mathcal{A}}) \in Q[\mathcal{R}] \Bigg\} \\
&\quad = \sum \Bigg\{ \sum \Big\{ \boldsymbol{\Delta}_w[q_{\mathcal{P}}, q''_{\mathcal{P}}] \cdot \boldsymbol{\Delta}_a[q''_{\mathcal{P}}, q'_{\mathcal{P}}] \;\Big|\; q_{\mathcal{A}} \overset{w}{\rightsquigarrow} q''_{\mathcal{A}}, \\
&\qquad\qquad q''_{\mathcal{A}} \overset{a}{\to} q'_{\mathcal{A}}, a \in \Sigma, w \in \Sigma^n \Big\} \;\Bigg|\; (q''_{\mathcal{P}}, q''_{\mathcal{A}}) \in Q[\mathcal{R}] \Bigg\} \\
&\quad = \sum \Big\{ \boldsymbol{\Delta}_{w'}[q_{\mathcal{P}}, q'_{\mathcal{P}}] \;\Big|\; q_{\mathcal{A}} \overset{w'}{\rightsquigarrow} q'_{\mathcal{A}}, w' \in \Sigma^{n+1} \Big\}.
\end{aligned}
$$

Since $\boldsymbol{E}^* = \sum_{i=0}^{\infty} \boldsymbol{E}^i$, Equation (4.1) follows. Using the matrix $\boldsymbol{E}^*$, it remains to compute $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}))$ as

$$\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A})) = \boldsymbol{\alpha}_{\mathcal{R}}^{\top} \cdot \boldsymbol{E}^* \cdot \boldsymbol{\gamma}_{\mathcal{R}}. \qquad \square$$

4.3 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 $n \in \omega$, 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 [159].

Lemma 4.3.1. *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \omega$, 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 4.2.1) 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 $\mathcal{L}(\mathcal{A}) = \Sigma^*$, 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 $\mathcal{L}(\mathcal{A}) = \Sigma^*$ 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

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 $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}') \leq \epsilon$

```
V := ∅;
for q ∈ Q in the order ⪯_{A,label(A,P)} do
    V := V ∪ {q}; A' := reduce(A, V);
    if |A'| ≤ n then break ;
return A', ε = error(A, V, label(A, P));
```

in Section 4.1). 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 $\{\epsilon\} \cup \Sigma \subseteq \mathcal{L}(\mathcal{A})$. Since the probability of any word from $\{\epsilon\} \cup \Sigma \subseteq \mathcal{L}(\mathcal{A})$ in $\mathcal{P}_{Exp}$ is non-zero, 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 $\mathcal{L}(\mathcal{A}') = \Sigma^*$, it holds that $d_{\mathcal{P}_{Exp}}(\mathcal{A}, \mathcal{A}') \leq 0$ iff $\mathcal{L}(\mathcal{A}) = \Sigma^*$. □

The notions defined above do not distinguish between introducing a *false positive* ($\mathcal{A}'$ accepts a word $w \notin \mathcal{L}(\mathcal{A})$) or a *false negative* ($\mathcal{A}'$ rejects a word $w \in \mathcal{L}(\mathcal{A})$) answers. To this end, we define *over-approximating* and *under-approximating* reductions as reductions for which the conditions $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ and $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{A}')$ 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 the restriction on $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}')$). Obviously, this approach is computationally infeasible.

4.4 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.

4.4.1 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 w.r.t. 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, $\mathit{error}(\mathcal{A}, V, \mathit{label}(\mathcal{A}, \mathcal{P}))$ estimates the error introduced by the application of $\mathit{reduce}(\mathcal{A}, V)$, 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 $\preceq_{\mathcal{A},\mathit{label}(\mathcal{A},\mathcal{P})}$ that defines which states of $\mathcal{A}$ are processed first and which are processed later. The ordering may take into account the precomputed labelling. The algorithm accepts an NFA $\mathcal{A}$, a PA $\mathcal{P}$, and $n \in \omega$ 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 $\preceq_{\mathcal{A},\mathit{label}(\mathcal{A},\mathcal{P})}$, until the size of the result of $\mathit{reduce}(\mathcal{A}, V)$ has reached the desired bound n (in our setting, *reduce* is always antitone, i.e., for $V \subseteq V'$, it holds that $|\mathit{reduce}(\mathcal{A}, V)| \geq |\mathit{reduce}(\mathcal{A}, V')|$). 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) $\mathit{error}(\mathcal{A}, V, \mathit{label}(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, \mathit{reduce}(\mathcal{A}, V))$,

(b) $|\mathit{reduce}(\mathcal{A}, Q[\mathcal{A}])| \leq 1$, *and*

(c) $\mathit{reduce}(\mathcal{A}, \emptyset) = \mathcal{A}$.

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 4.4.1. *Algorithm 1 is correct if **C1** holds.*

Proof. Follows straightforwardly from Condition **C1**. □

4.4.2 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 4.4.2. *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. $d_{\mathcal{P}}(\mathcal{A}, \mathcal{A}') \leq \epsilon$

$\ell := \mathit{label}(\mathcal{A}, \mathcal{P})$;
$V := \emptyset$;
for $q \in Q$ *in the order* $\preceq_{\mathcal{A},\mathit{label}(\mathcal{A},\mathcal{P})}$ **do**
  $e := \mathit{error}(\mathcal{A}, V \cup \{q\}, \ell)$;
  if $e \leq \epsilon$ **then** $V := V \cup \{q\}$;
return $\mathcal{A}' = \mathit{reduce}(\mathcal{A}, V)$;

4.4.3 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 $Q \setminus R$, followed by removing useless states, to construct a reduced automaton $\mathcal{A}' = \mathit{trim}(\mathcal{A}_{|Q\setminus R})$. Note that the natural decision problem corresponding to this reduction is also **PSPACE**-complete.

Lemma 4.4.3. *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \omega$, 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 4.2.1) 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. $\mathcal{L}(\mathcal{A}') = x^*.\mathcal{L}(\mathcal{A})$. $\mathcal{A}'$ 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

$$\Pr_{\mathcal{P}}(w) = \begin{cases} \mu^{|w'|+1} & \text{for } w = x^n.w', w' \in \Sigma^*, \text{and } \mu = \frac{1}{|\Sigma|+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

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 $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{B})) = 1$. Using the automata defined above, we construct an NFA $\mathcal{C} = \mathcal{A}' \uplus \mathcal{B}$. NFA $\mathcal{C}$ has $2n$ states, the language of $\mathcal{C}$ is $\mathcal{L}(\mathcal{C}) = x^*.\mathcal{L}(\mathcal{A}) \cup x^n.\Sigma^*$ and its probability is $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{C})) = 1$.

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 $\mathcal{L}(\mathcal{A}) = \Sigma^*$. 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, $\mathcal{L}(\mathcal{C}_{|R}) \subseteq x^*.\mathcal{L}(\mathcal{A})$. Because $d_{\mathcal{P}}(\mathcal{C}, \mathcal{C}_{|R}) \leq 0$, we know that $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{C}_{|R})) = 1$, so $x^n.\Sigma^* \subseteq x^*.\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{C}_{|R})$

and, therefore, $\mathcal{L}(\mathcal{A}) = \Sigma^*$. For the other direction, if $\mathcal{L}(\mathcal{A}) = \Sigma^*$, 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 4.4.3 shows that the pruning reduction is as hard as a general reduction (cf. Lemma 4.3.1), the pruning reduction is more amenable to using heuristics like the greedy algorithms from Section 4.4.1 and Section 4.4.2. We instantiate *reduce*, *error*, and *label* in these high-level algorithms in the following way (the subscript p stands for *pruning*):

$$\begin{aligned} \mathit{reduce}_p(\mathcal{A}, V) &= \mathit{trim}(\mathcal{A}_{|Q \setminus V}), \\ \mathit{error}_p(\mathcal{A}, V, \ell) &= \min_{V' \in \lfloor V \rfloor_p} \sum \left\{ \ell(q) \mid q \in V' \right\}, \end{aligned}$$

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 $\mathit{reduce}_p(\mathcal{A}, V)$ may, in general, yield the same automaton as $\mathit{reduce}_p(\mathcal{A}, V')$. 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 $\mathit{reduce}_p(\mathcal{A}, V_1) = \mathit{reduce}_p(A, V_2)$ 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 (w.r.t. $\sqsubseteq_p$). The value of the approximation $\mathit{error}_p(A, V, \ell)$ is therefore the minimum of the sum of errors over all sets from $\lfloor V \rfloor_p$.

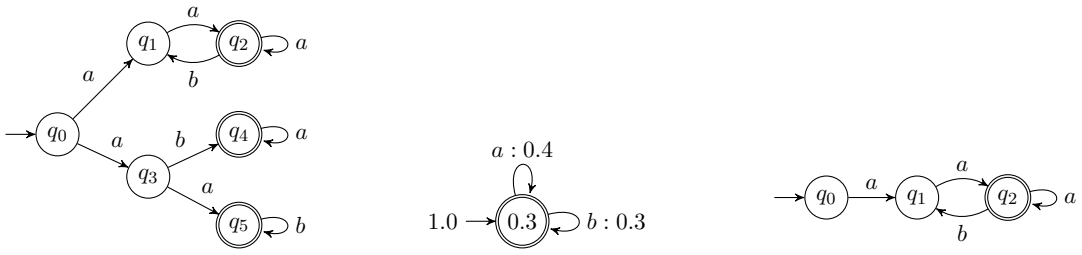
Note that the size of $\lfloor V \rfloor_p$ can again be exponential, and thus we employ a greedy approach for guessing an optimal V'. Clearly, this cannot affect the soundness of the algorithm, but only decreases the precision of the bound on the distance. Our experiments indicate that for automata 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_p^1, label_p^2, and label_p^3. Given an NFA $\mathcal{A}$ and a PA $\mathcal{P}$, they generate the labellings ℓ_p^1, ℓ_p^2, and ℓ_p^3, respectively, defined as

$$\begin{aligned} \ell_p^1(q) &= \sum \left\{ \Pr_{\mathcal{P}}(\mathcal{L}_{\mathcal{A}}^{\flat}(q')) \;\middle|\; q' \in \mathit{reach}(\{q\}) \cap F \right\}, \\ \ell_p^2(q) &= \Pr_{\mathcal{P}}\left(\mathcal{L}_{\mathcal{A}}^{\flat}(F \cap \mathit{reach}(q)) \right), \\ \ell_p^3(q) &= \Pr_{\mathcal{P}}\left(\mathcal{L}_{\mathcal{A}}^{\flat}(q).\mathcal{L}_{\mathcal{A}}(q) \right). \end{aligned}$$

A state label $\ell(q)$ approximates the error of the words removed from $\mathcal{L}(\mathcal{A})$ when q is removed. More concretely, $\ell_p^1(q)$ 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 $\ell_p^1(q)$ (i) counts the error of a word accepted in two different final states of $\mathit{reach}(q)$ twice and (ii) it also considers words that are accepted in some final state in $\mathit{reach}(q)$ without going through q. The labelling ℓ_p^2 deals with (i) by computing the total probability of the banguage of the set of all final states reachable from q, and the labelling ℓ_p^3 in addition also deals with (ii) by only considering words that traverse through q (they can, however, be accepted in some final state not in $\mathit{reach}(q)$ by a run completely disjoint from q and $\mathit{reach}(q) \cap F$, so even ℓ_p^3 can still be imprecise). Note that if $\mathcal{A}$ is unambiguous, then $\ell_p^1 = \ell_p^2$.

Each state labelling is given as the probability (or the sum of probabilities in the case of ℓ_p^1) of the language related to q. Therefore, when computing the particular label of



(a) NFA $\mathcal{A}$ **(b)** PA $\mathcal{P}$ **(c)** NFA $\mathcal{A}'$

Figure 4.2: An example of an NFA $\mathcal{A}$ and a PA $\mathcal{P}$ over $\{a, b\}$ as an input for the pruning reduction with a resulting NFA $\mathcal{A}'$.

q, we first modify $\mathcal{A}$ to obtain $\mathcal{A}'$ accepting the language related to the labelling. Then, we compute the value of $\Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}'))$ using the algorithm from Section 4.2. 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 RE 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 example shows the pruning reduction of a simple NFA.

Example 4.4.1. *Consider NFA $\mathcal{A}$ and PA $\mathcal{P}$ from Figures 4.2a and 4.2b, respectively. State labels are given as*

	q_0	q_1	q_2	q_3	q_4	q_5
ℓ_p^1	*0.2285*	*0.1*	*0.1*	*0.1285*	*0.06*	*0.0685*
ℓ_p^2	*0.1805*	*0.1*	*0.1*	*0.1285*	*0.06*	*0.0685*
ℓ_p^3	*0.1805*	*0.1*	*0.1*	*0.1285*	*0.06*	*0.0685*

Let us look more deeply for instance on labels of q_0. The value of $\ell_p^1(q_0)$ is given by $\ell_p^1(q_0) = \Pr_{\mathcal{P}}(\mathcal{L}_{\mathcal{A}}^{\flat}(q_2)) + \Pr_{\mathcal{P}}(\mathcal{L}_{\mathcal{A}}^{\flat}(q_4)) + \Pr_{\mathcal{P}}(\mathcal{L}_{\mathcal{A}}^{\flat}(q_5)) \approx 0.1 + 0.06 + 0.0685 = 0.2285$. Since $\mathcal{L}_{\mathcal{A}}^{\flat}(q_5) \cap \mathcal{L}_{\mathcal{A}}^{\flat}(q_2) = \{aa\}$, we have $\ell_p^2(q_0) = \ell_p^3(q_0) = \ell_p^1(q_0) - \Pr_{\mathcal{P}}(\{aa\}) \approx 0.1805$.

Based on the state labels, the pruning reduction of $\mathcal{A}$ with the error bound $\epsilon = 0.15$ and the state order given by values of ℓ_p^3 leads to the NFA in Figure 4.2c. The set V in Algorithm 2 after the for loop contains states $V = \{q_3, q_4, q_5\}$ with the corresponding error estimation $error_p(\mathcal{A}, V, \ell_p^1) = 0.1285$ (in this case a choice of a state label does not affect the result). The part of $\mathcal{A}$ corresponding to states V was removed because it is less probable to reach a final state in this part than in the part corresponding to states $\{q_0, q_1, q_2\}$.

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_p^1, ℓ_p^2, and ℓ_p^3.

Lemma 4.4.4. *For every $x \in \{1, 2, 3\}$, the functions $reduce_p$, $error_p$, and $label_p^x$ satisfy **C1**. Moreover, consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, and let $\ell_p^x = label_p^x(\mathcal{A}, \mathcal{P})$ for $x \in \{1, 2, 3\}$. Then, for each $q \in Q[\mathcal{A}]$, we have $\ell_p^1(q) \geq \ell_p^2(q) \geq \ell_p^3(q)$.*

Proof. We start by proving the inequalities $\ell_p^1(q) \geq \ell_p^2(q) \geq \ell_p^3(q)$ 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_p^1, we may

sum probabilities of the same words multiple times. The second inequality follows from the inclusion $\mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \subseteq \mathcal{L}^{\flat}_{\mathcal{A}}(F \cap \mathit{reach}(q))$.

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

$$\mathit{error}_p(\mathcal{A}, V, \mathit{label}^x_p(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, \mathit{reduce}_p(\mathcal{A}, V)),$$

  we prove it for $\ell^3_p = \mathit{label}^3_p(\mathcal{A}, \mathcal{P})$; 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

$$\begin{aligned}
\mathcal{L}(\mathcal{A}) \mathbin{\triangle} \mathcal{L}(\mathit{reduce}_p(\mathcal{A}, V)) &= \mathcal{L}(\mathcal{A}) \mathbin{\triangle} \mathcal{L}\left(\mathit{reduce}_p(\mathcal{A}, V')\right) && \text{⟅def. of } \sqsubseteq_p\text{⟆} \\
&= \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}\left(\mathit{reduce}_p(\mathcal{A}, V')\right) && \text{⟅}\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathit{reduce}_p(\mathcal{A}, V'))\text{⟆} \\
&\subseteq \bigcup_{q \in V'} \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q). && \text{⟅def. of } \mathit{reduce}_p\text{⟆}
\end{aligned} \tag{4.4}$$

  Finally, using (4.4), we obtain

$$\begin{aligned}
d_{\mathcal{P}}(\mathcal{A}, \mathit{reduce}_p(\mathcal{A}, V)) &= \Pr_{\mathcal{P}}(\mathcal{L}(\mathcal{A}) \mathbin{\triangle} \mathcal{L}\left(\mathit{reduce}_p(\mathcal{A}, V')\right)) && \text{⟅def. of } d_{\mathcal{P}}\text{⟆} \\
&\leq \sum_{q \in V'} \Pr_{\mathcal{P}}(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q)) && \text{⟅(4.4)⟆} \\
&= \sum\{\ell^3_p(q) \mid q \in V'\} && \text{⟅def. of } \ell^3_p\text{⟆} \\
&= \min_{V'' \in \lfloor V \rfloor_p} \sum\{\ell^3_p(q) \mid q \in V''\} && \text{⟅def. of } V'\text{⟆} \\
&= \mathit{error}_p(\mathcal{A}, V, \ell^3_p). && \text{⟅def. of } \mathit{error}_p\text{⟆}
\end{aligned}$$

- **C1**(b): $|\mathit{reduce}_p(\mathcal{A}, Q[\mathcal{A}])| \leq 1$ because

$$|\mathit{reduce}_p(\mathcal{A}, Q[\mathcal{A}])| = |\mathit{trim}(\mathcal{A}_{|\emptyset})| = 0.$$

- **C1**(c): $\mathit{reduce}_p(\mathcal{A}, \emptyset) = \mathcal{A}$ since

$$\mathit{reduce}_p(\mathcal{A}, \emptyset) = \mathit{trim}(\mathcal{A}_{|Q[\mathcal{A}]}) = \mathcal{A}$$

  (we assume that $\mathcal{A}$ is trimmed at the input). □

4.4.4 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 $\mathit{sl}(\mathcal{A}, R)$ 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 4.4.5. *Consider an NFA $\mathcal{A}$, a PA $\mathcal{P}$, a bound on the number of states $n \in \omega$, 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 we did in the proof of Lemma 4.4.3.

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 $\mathcal{L}(\mathcal{A}) = \Sigma^*$ 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*):

$$\begin{aligned} \mathit{reduce}_{sl}(\mathcal{A}, V) &= \mathit{trim}(sl(\mathcal{A}, V)), \\ \mathit{error}_{sl}(\mathcal{A}, V, \ell) &= \sum \{\ell(q) \mid q \in \min(\lfloor V \rfloor_{sl})\}, \end{aligned}$$

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 $\mathit{label}^1_{sl}, \mathit{label}^2_{sl}$, 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:

$$\begin{aligned} \ell^1_{sl}(q) &= \mathit{weight}_{\mathcal{P}}(\mathcal{L}^{\flat}_{\mathcal{A}}(q)), \\ \ell^2_{sl}(q) &= \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^*\right), \\ \ell^3_{sl}(q) &= \ell^2_{sl}(q) - \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q)\right). \end{aligned}$$

In the definitions above, the function $\mathit{weight}_{\mathcal{P}}(w)$ for a PA $\mathcal{P} = (\boldsymbol{\alpha}, \boldsymbol{\gamma}, \{\boldsymbol{\Delta}_a\}_{a \in \Sigma})$ and a word $w \in \Sigma^*$ is defined as $\mathit{weight}_{\mathcal{P}}(w) = \boldsymbol{\alpha}^{\top} \cdot \boldsymbol{\Delta}_w \cdot \mathbf{1}$ (i.e., similarly as $\Pr_{\mathcal{P}}(w)$ but with the final weights $\boldsymbol{\gamma}$ discarded), and $\mathit{weight}_{\mathcal{P}}(L)$ for $L \subseteq \Sigma^*$ is defined as $\mathit{weight}_{\mathcal{P}}(L) = \sum_{w \in L} \mathit{weight}_{\mathcal{P}}(w)$.

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: (i) the probability of some words may be included twice, e.g., when $\mathcal{L}^{\flat}_{\mathcal{A}}(q) = \{a, ab\}$, the probabilities of all words from $\{ab\}.\Sigma^*$ are included twice in $\ell^1_{sl}(q)$ because $\{ab\}.\Sigma^* \subseteq \{a\}.\Sigma^*$, and (ii) $\ell^1_{sl}(q)$ can also contain probabilities of words already accepted on a run traversing q. The state labelling ℓ^2_{sl} deals with (i) by considering the probability of the language $\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^*$, and ℓ^3_{sl} deals also with (ii) by subtracting from the result of ℓ^2_{sl} the probabilities of the words that pass through q and are accepted.

Example 4.4.2. *Consider NFA $\mathcal{A}$ and PA $\mathcal{P}$ from Figures 4.2a and 4.2b, respectively. State labels are given as*

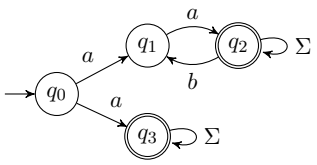


Figure 4.3: A resulting NFA after the self-loop reduction.

	q_0	q_1	q_2	q_3	q_4	q_5
ℓ^1_{sl}	*1*	*0.5*	*0.33*	*0.4*	*0.2*	*0.2285*
ℓ^2_{sl}	*1*	*0.4*	*0.16*	*0.4*	*0.12*	*0.16*
ℓ^3_{sl}	*0.8195*	*0.3*	*0.06*	*0.2715*	*0.06*	*0.0915*

Let us look more profoundly for instance on labels of q_1*. The value of* $\ell^1_{sl}(q_1)$ *is given by* $\ell^1_{sl}(q_1) = weight_{\mathcal{P}}(\mathcal{L}^{\flat}_{\mathcal{A}}(q_1)) = 0.5$*. The value of* $\ell^2_{sl}(q_1)$ *is given by* $\ell^2_{sl}(q_1) = \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q_1).\Sigma^*\right) = \Pr_{\mathcal{P}}\left(a.\Sigma^*\right) = 0.4$*. Finally,* $\ell^3_{sl}(q_1) = 0.4 - \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q_1).\mathcal{L}_{\mathcal{A}}(q_1)\right) = 0.3$*.*

Based on the state labels ℓ^3_{sl}*, the self-loop reduction of* $\mathcal{A}$ *with the size bound* $n = 4$ *and the state order given by values of* ℓ^3_{sl} *leads to the NFA in Figure 4.3. The set* V *in Algorithm 1 after the for loop contains states* $V = \{q_2, q_3, q_4, q_5\}$ *with the corresponding error estimation* $error_{sl}(\mathcal{A}, V, \ell^3_{sl}) = 0.3315$*. Note that this automaton can be further reduced by language-preserving reductions.*

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. Section 4.4.3). 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 4.4.6. *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

$$\sum_{w \in \mathcal{L}^{\flat}_{\mathcal{A}}(q)} weight_{\mathcal{P}}(w) = \sum_{w \in \mathcal{L}^{\flat}_{\mathcal{A}}(q)} \Pr_{\mathcal{P}}(w.\Sigma^*),$$

which, in turn, implies

$$\ell^1_{sl}(q) = weight_{\mathcal{P}}(\mathcal{L}^{\flat}_{\mathcal{A}}(q)) = \sum_{w \in \mathcal{L}^{\flat}_{\mathcal{A}}(q)} \Pr_{\mathcal{P}}\left(w.\Sigma^*\right) \geq \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^*\right) = \ell^2_{sl}(q). \tag{4.5}$$

For example, for $\mathcal{L}^{\flat}_{\mathcal{A}}(q) = \{w, wa\}$ where $w \in \Sigma^*$ and $a \in \Sigma$, we have

$$\begin{aligned} weight_{\mathcal{P}}(\mathcal{L}^{\flat}_{\mathcal{A}}(q)) &= weight_{\mathcal{P}}(\{w, wa\}) = weight_{\mathcal{P}}(w) + weight_{\mathcal{P}}(wa) \\ &= \Pr_{\mathcal{P}}(w.\Sigma^*) + \Pr_{\mathcal{P}}(wa.\Sigma^*), \end{aligned} \tag{4.6}$$

while

$$\Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^*\right) = \Pr_{\mathcal{P}}\left(\{w, wa\}.\Sigma^*\right) = \Pr_{\mathcal{P}}\left(w.\Sigma^*\right).$$

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 $\mathit{error}_{sl}(\mathcal{A}, V, \mathit{label}^x_{sl}(\mathcal{A}, \mathcal{P})) \geq d_{\mathcal{P}}(\mathcal{A}, \mathit{reduce}_{sl}(\mathcal{A}, V))$, we prove that the inequality holds for $\ell^3_{sl} = \mathit{label}^3_{sl}(\mathcal{A}, \mathcal{P})$; 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

$$\begin{aligned}
&\mathcal{L}(\mathcal{A}) \triangle \mathcal{L}(\mathit{reduce}_{sl}(\mathcal{A}, V)) \\
&\quad = \mathcal{L}(\mathcal{A}) \triangle \mathcal{L}\left(\mathit{reduce}_{sl}(\mathcal{A}, V')\right) && \text{\{def. of } \sqsubseteq_{sl}\text{\}} \\
&\quad = \mathcal{L}\left(\mathit{reduce}_{sl}(\mathcal{A}, V')\right) \setminus \mathcal{L}(\mathcal{A}) && \text{\{}\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathit{reduce}_{sl}(\mathcal{A}, V'))\text{\}} \qquad (4.7) \\
&\quad \subseteq \bigcup_{q \in V'} \mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \bigcup_{q \in V'} \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q). && \text{\{def. of } \mathit{reduce}_{sl}\text{\}}
\end{aligned}$$

  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 $\mathit{reduce}_{sl}(\mathcal{A}, V)$ as

$$\begin{aligned}
&d_{\mathcal{P}}(\mathcal{A}, \mathit{reduce}_{sl}(\mathcal{A}, V)) \\
&\quad \leq \Pr_{\mathcal{P}}\left(\bigcup_{q \in V'} \mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \bigcup_{q \in V'} \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \right) && \text{\{(4.7)\}} \\
&\quad \leq \Pr_{\mathcal{P}}\left(\bigcup_{q \in V'} \left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \right) \right) \\
&\quad \text{\{properties of union and set difference\}} \\
&\quad \leq \sum_{q \in V'} \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^* \setminus \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \right) && \text{\{union bound\}} \\
&\quad = \sum_{q \in V'} \left(\Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^* \right) - \Pr_{\mathcal{P}}\left(\mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \right) \right) \\
&\quad \text{\{prop. of Pr and the fact that } \mathcal{L}^{\flat}_{\mathcal{A}}(q).\mathcal{L}_{\mathcal{A}}(q) \subseteq \mathcal{L}^{\flat}_{\mathcal{A}}(q).\Sigma^*\text{\}} \\
&\quad = \sum \{ \ell^3_{sl}(q) \mid q \in \min(\lfloor V \rfloor_{sl}) \} && \text{\{def. of } \ell^3_{sl} \text{ and } V'\text{\}} \\
&\quad = \mathit{error}_{sl}(\mathcal{A}, V, \ell^3_{sl}).
\end{aligned}$$

- **C1**(b): $|\mathit{reduce}_{sl}(\mathcal{A}, Q[\mathcal{A}])| \leq 1$ because, from the definition, $|\mathit{reduce}_{sl}(\mathcal{A}, Q[\mathcal{A}])| = |\mathit{trim}(sl(\mathcal{A}, Q[\mathcal{A}]))| \leq 1$.

- **C1**(c): $\mathit{reduce}_{sl}(\mathcal{A}, \emptyset) = \mathcal{A}$ since

$$\mathit{reduce}_{sl}(\mathcal{A}, \emptyset) = \mathit{trim}(sl(\mathcal{A}, \emptyset)) = \mathcal{A}$$

  (we assume that $\mathcal{A}$ is trimmed at the input). □

4.5 Experimental Evaluation

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 [269], a popular open source NIDS. The version of Appreal used for the evaluation in this chapter is available as an artifact [282] for the TACAS'18 artifact virtual machine [136].

4.5.1 Network Traffic Model

The reduction we describe in this chapter is driven by a probabilistic model representing a distribution over the words from Σ^*, and the formal guarantees are also w.r.t. 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 non-zero 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 [71, 270]. Indeed, we experimented with the use of Alergia [71], 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 Section 4.5.2, 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.

[6] `https://github.com/vhavlena/appreal/tree/tacas18`

Table 4.1: Results for the `http-malicious` RE, $|\mathcal{A}_{\texttt{mal}}| = 249$, $|\mathcal{A}^{\text{RED}}_{\texttt{mal}}| = 98$, $time(\text{REDUCE}) = 3.5\,\text{s}$, $time(label^2_{sl}) = 38.7\,\text{s}$, $time(\text{Exact}) = 3.8\text{–}6.5\,\text{s}$, and $LUTs(\mathcal{A}^{\text{RED}}_{\texttt{mal}}) = 382$.

(a) size-driven reduction

k	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{mal}}\rvert$	$\lvert\mathcal{A}'_{\texttt{mal}}\rvert$	Error bound	Exact error	Traffic error	LUTs
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

(b) error-driven reduction

ϵ	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{mal}}\rvert$	$\lvert\mathcal{A}'_{\texttt{mal}}\rvert$	Error bound	Exact error	Traffic error
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

4.5.2 Evaluation

We start this section by introducing the experimental setting, namely, the integration of our reduction techniques into the tool chain implementing efficient RE 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 [234] to transform them into a single NFA $\mathcal{A}$. Before applying APPREAL, we use the state-of-the-art NFA reduction tool REDUCE [208] to reduce $\mathcal{A}$. REDUCE performs a language-preserving reduction of $\mathcal{A}$ using advanced variants of simulation [206] (in the experiment reported in Table 4.3, 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 Section 4.4 w.r.t. 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* (Section 4.4.4). 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* (Section 4.4.1) 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 Section 4.4.1 and Section 4.4.2 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 REs 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 ($time(\text{Reduce})$). 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 ($time(\text{Exact})$), 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 Section 4.5.3). The meaning of the columns in the tables is the following:

k/ϵ is the parameter of the reduction. In particular, $\boldsymbol{k}$ is used for the size-driven reduction and denotes the desired reduction ratio $k = \frac{n}{|\mathcal{A}^{\text{Red}}|}$ 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.

$|\mathcal{A}^{\text{App}}|$ 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).

$|\mathcal{A}'|$ 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 Section 4.4.

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 4.3 (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.

Approximation errors

Table 4.1 presents the results of the self-loop reduction for the NFA $\mathcal{A}_{\texttt{mal}}$ describing REs from `http-malicious`. We can observe that the differences between the upper bounds on

Table 4.2: Results for the `http-attacks` RE, size-driven reduction, $|\mathcal{A}_{\texttt{att}}| = 142$, $|\mathcal{A}^{\text{RED}}_{\texttt{att}}| = 112$, $time(\text{REDUCE}) = 7.9\,\text{s}$, $time(label^2_{sl}) = 28.3\,\text{min}$, $time(\text{Exact}) = 14.0\text{–}16.4\,\text{min}$.

k	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{att}}\rvert$	$\lvert\mathcal{A}'_{\texttt{att}}\rvert$	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

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 4.1b shows that a significant reduction is obtained when the error threshold ϵ is increased from 0.04 to 0.07.

Table 4.2 presents the results of the size-driven self-loop reduction for NFA $\mathcal{A}_{\texttt{att}}$ describing `http-attacks` REs. We can observe that the error bounds provide again a very good approximation of the real probabilistic distance. 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 $\mathcal{A}^{\text{RED}}_{\texttt{att}}$. 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 4.3 (presenting the results for $\mathcal{A}_{\texttt{bd}}$ constructed from `http-backdoor` REs) 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 5-fold reduction is obtained for the traffic error 0.03 % and a 10-fold reduction is obtained for the traffic error 6.3 %. We discuss the practical impact of such a reduction in Section 4.5.3.

Performance of the approximate reduction

In all our experiments (Tables 4.1 and 4.2), 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. Section 4.4.4). 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.

Table 4.3: Results for `http-backdoor`, size-driven reduction, $|\mathcal{A}_{\texttt{bd}}| = 1,352$, $time(label^2_{sl}) = 19.9\,\text{min}$, $LUTs(\mathcal{A}^{\text{RED}}_{\texttt{bd}}) = 2,266$.

k	$\lvert\mathcal{A}^{\text{APP}}_{\texttt{bd}}\rvert$	$\lvert\mathcal{A}'_{\texttt{bd}}\rvert$	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	1063
0.5	676 (1.3 m)	473 (41.8 s)	1.2e-17	0.0	1249
0.7	946 (1.4 m)	739 (2.1 m)	8.3e-30	0.0	1735
0.9	1216 (1.5 m)	983 (5.6 m)	1.3e-52	0.0	2033

As explained in Section 4.4, 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 behavior 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 behavior 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 min 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 4.3; 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.

4.5.3 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 [204] implementing a high-speed NIDS pre-filter in an FPGA. The crucial challenge is to obtain a pre-filter that has sufficiently small false positive rate (and no false negative) while it can handle the traffic of current networks operating on 100 Gbps and beyond. The implementation of NFAs performing the RE matching in FPGAs uses two types of HW resources: LUTs which are use to build the combinational circuit representing the NFA transition function and flip-flops representing the 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 RE 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 [204], the engine containing a single unit (i.e. the single NFA

implementation) can achieve the throughput of 6.4 Gbps (200 MHz × 32 b). Therefore, 16 units are required for the desired link speed of 100 Gbp 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 (15000/16) for 100 Gbps and 238 LUTs for 400 Gbps, respectively. These bounds directly limit the complexity of regular expressions the engine can handle.

We now analyze the resource consumption of the matching engine for two automata, `http-backdoor` ($\mathcal{A}^{\text{RED}}_{\texttt{bd}}$) and `http-malicious` ($\mathcal{A}^{\text{RED}}_{\texttt{mal}}$), and evaluate the impact of the reduction techniques. Recall that the automata represent two important sets of know network attacks from SNORT [269].

- **100 Gbps**: For this speed, $\mathcal{A}^{\text{RED}}_{\texttt{mal}}$ 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, $\mathcal{A}^{\text{RED}}_{\texttt{bd}}$ 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 4.3 shows that using our framework, we are able to reduce $\mathcal{A}^{\text{RED}}_{\texttt{bd}}$ such that it uses 894 LUTs (for $\boldsymbol{k} = 0.3$), and so all the needed 16 units fit into the FPGA, yielding the throughput over 100 Gbps and the theoretical error bound of a false positive $\leq 3.4 \times 10^{-8}$ w.r.t. the network traffic model $\mathcal{P}_{HTTP}$.

- **400 Gbps**: RE matching at this speed is extremely challenging. In the case of $\mathcal{A}^{\text{RED}}_{\texttt{bd}}$, the reduction $\boldsymbol{k} = 0.1$ is required to fit 63 units in the available space. As such reduction has error bound almost 1, this solution is not very useful due to a prohibitively high false positive rate. The situation is better for $\mathcal{A}^{\text{RED}}_{\texttt{mal}}$. 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 throughput over 400 Gbps with the theoretical error bound of a false positive $\leq 8.7 \times 10^{-8}$ w.r.t. the model $\mathcal{P}_{HTTP}$.

4.6 Conclusion

In this chapter, 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 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 REs of malicious traffic where it could not be applied before. This chapter is based on the work published in the proceedings of TACAS'18 [283] and in the STTT journal [307].

One direction of a further research could include development of better (automatic) ways of learning of a sufficiently precise and concise probabilistic model of the input traffic. Another direction could focus on other possible reductions of NFAs, for instance reductions based on merging of states maintaining the proposed formal guarantees.

Chapter 5

Lightweight Approximate Reduction of NFAs

As we said in Chapters 3 and 4, NFA-based RE pattern matching in high-speed networks is a demanding task for which hardware acceleration is a suitable solution. Recall also that a HW unit is usually based on the FPGA technology and that it serves as a pre-filter of input traffic sending suspicious packets for further inspection to software. The resources of FPGAs are restricted and, therefore, the automata representing REs of interest need to be as small as possible.

In this chapter, we build upon the results from Chapter 4 where we proposed approximate reduction of NFAs with formal guarantees. The disadvantages of that approach are (i) its complexity (recall that even the computation of the state labelling is **PSPACE**-complete), and (ii) the requirement on a model of the network traffic. In particular, (i) is a serious issue when dealing with large automata that occur in RE matching of network traffic. For this reason, we propose *lightweight* approximate reduction, *over-approximating* the language, suitable for a reduction of large NFAs. In contrast to the approaches from Chapter 4, the approximate reductions presented in this chapter are steered not by a probabilistic model of the traffic but directly by a multiset of packets representing a typical traffic. The price we pay for this lightweightness is a sacrifice of guarantees that we can no longer provide (we do not work with a model of the network traffic anymore). We can just measure the number of misclassified packets from the sample. Recall that the reductions from the previous chapter can provide formal guarantees w.r.t. a probabilistic traffic model. Since Chapter 4 is closer to a theoretical proof-of-concept, we, in this chapter, focus on more practical aspects of the problem including a more detailed evaluation of the synthesized architectures for high-speed pattern matching.

Overview of the proposed approach. Our approximate NFA reduction takes an advantage of particularities of standard network traffic. Namely, given an NFA constructed from the REs of interest, we label its states with their *significance*—the likelihood that they will be used during processing a packet—, and then simplify the least significant parts of the automaton. The simplification is implemented by *pruning* and *merging* of the insignificant states. The significance of a state is determined using *training traffic*, a finite sample of "standard" traffic from the network node where the NIDS is to be deployed (it may be necessary to generate a new design once in a while). The reduction scales well—the worst-case time complexity of the most expensive step, computing the state labelling,

is $\mathcal{O}(n^2k)$ where n is the number of states of the NFA and k is the size of the training traffic (since the automata are usually sparse, the quadratic factor is rarely an issue on real-world examples).

We implemented the proposed approach and evaluated it on REs taken from the NIDS Snort and other resources. We were able to obtain a substantial reduction of the size of the NFAs while keeping the number of false positives low. Based on our approximate reduction we were able to perform RE matching at 100 and 400 Gbps on sets of large REs.

Related work. Many different architectures for resource-efficient mapping of NFAs for fast RE matching into FPGAs have been designed (see Section 3.2, paragraph related to NFA-based acceleration) as well as techniques for reduction of automata (see Section 3.3).

We mention only the most relevant works related to high-speed pattern matching. To increase the RE matching speed, some architectures make the NFAs process multiple bytes of the input per clock cycle [254]. Achieving the 100 Gbps throughput, which requires more than 64 bytes to be processed at once is, however, not possible because of frequency issues. In [36] a multi-striding technique, which is widely used to increase the throughput of many NFA-based RE matching architectures, was introduced. Multi-striding alone, however, cannot increase the processing speed to 100 Gbps because with the length of the stride, the NFA grows rapidly, and the frequency drops dramatically [204].

In this chapter, we use the architecture of [204] introducing parallel pipelined automata, which can scale the throughput of NFA-based RE matching to over 100 Gbps. In a recent work, a novel FPGA architecture that significantly improved the throughput of DFA-based RE matching was introduced [295]. The architecture achieves a throughput of 140 Gbps on a single FPGA chip for NIDS modules where the underlying *DFAs* have up to 10k states (corresponding to 34 REs of Snort). We can achieve, on a similar chip, a throughput beyond 200 Gbps for much more complicated sets of REs (e.g., Snort's `spyware` module with 461 REs where the underlying NFA has ∼10k states and the corresponding DFA is prohibitively large—our attempt of its determinization depleted the available memory (32 GiB) after reaching 616k states).

The work closest to our NFA reduction techniques is [194]. In [194], the authors address the issue of software-based acceleration of matching REs describing network attacks in Snort. To reduce the number k of membership tests needed for matching a packet against k distinct DFAs, [194] builds a "search tree" with the k DFAs in its leaves and with the inner nodes occupied by, preferably, small DFAs that over-approximate the union of their children (the precise DFA accepting the union is prohibitively large). Matching a packet then means to propagate it down the tree as long as it belongs to languages of the DFA nodes. The over-approximating DFAs are constructed using a similar notion of significance of states as in our approach. The differences from our work are the following: (i) [194] needs several membership tests per packet, (ii) it targets only software and its hardware implementation would be too complex, (iii) it does not consider reduction of NFAs, and (iv) it uses only a pruning-based reduction (we also employ merging and simulation-based reductions).

Finally, we compare our approach to RE-matching techniques that use modern general-purpose GPUs. As in FPGAs, we can distinguish architectures leveraging both DFAs and NFAs. Prominent GPU architectures based on DFAs include Gregex [287], hierarchical parallel machines [193], and a recent work employing algorithm/implementation co-optimization based on a GPU performance model [151]. These architectures are able to perform RE matching at the theoretical throughput of 100–150 Gbps. Their practical

performance is, however, limited by the packet transfer throughput (e.g., the throughput of [287] was 25 Gbps on NVIDIA GTX260), and, indeed, the complexity of RE modules they can handle due to determinization.

An RE matching architecture for GPUs based on NFAs was proposed in iNFAnt [72] and further improved in [20]. In contrast to the aforementioned DFA-based solutions, this architecture can handle complex RE modules where the underlying NFAs have over 10 thousand states. The performance of this work is, however, significantly inferior to our considered FPGA-based architecture. For example, for a category of SNORT modules where the underlying NFAs have 3–18 thousand states, [20] reports an overall throughput of 1–2 Gbps on NVIDIA Tesla c2050. Our experiments in Section 5.3 show that, on SNORT modules with similar sizes, our approach is able to achieve a throughput of over 100 Gbps.

More recently, there emerged approaches, complementary to our techniques, increasing throughput of NIDSes based on acceleration of exact string matching (as in [303], where, however, RE-matching itself is performed in SW). Another recent technique focuses on increasing the throughput of RE matching by a pre-filter of strings that are guaranteed to occur in a text matched by the REs [117].

Chapter outline. In Section 5.1, we briefly describe the considered HW-accelerated RE matching architecture based on FPGAs. In Section 5.2, we propose our lightweight approximation techniques. Finally, Section 5.3 deals with the experimental evaluation of the proposed approach and Section 5.4 concludes the chapter.

5.1 The FPGA Architecture

In this chapter, we use an architecture proposed in [204] involving *pipelined automata*. We use the NFA-based acceleration because, as already mentioned in Section 3.2, NFAs are much more succinct than DFAs and they can also be mapped into FPGAs more efficiently than DFAs.

The architecture based on pipelined automata uses k connected copies of automata $\mathcal{A}_0, \ldots, \mathcal{A}_{k-1}$ to achieve parallel processing of input packets. The automata are arranged in a ring topology and each automaton can send a current configuration (active states) of the matching to the next automaton in the ring. The automata read an input from a shared packet buffer (each NFA has its own row in the shared buffer where it reads input data). Symbols of an incoming packet payload are distributed across the buffer such that no data conflicts occur and, on the other hand, maximal parallel processing is achieved. Each copy of the NFA $\mathcal{A}_i$ computes successors based on the read data and the configuration obtained by the previous NFA $\mathcal{A}_{(i-1 \mod k)}$. If the reading of a packet is finished, the automata publish matched final states. Consider, for instance, a packet consisting of two parts p_1, p_2. In the first clock cycle, p_1 is processed by $\mathcal{A}_0$ and the configuration is passed to $\mathcal{A}_1$. In the second clock cycle, $\mathcal{A}_1$ processes p_2 but $\mathcal{A}_0$ can start reading the first part of the next packet. Using such pipelined automata, we can theoretically achieve parallel processing of k packets.

In order to achieve a throughput necessary for high-speed networks, a lot of copies of NFAs need to be synthesized in the FPGA chip. For instance, processing 100 Gbps input network traffic requires 64 concurrently functioning RE matching units (of 8-bit input width operating at 200 MHz) and 400 Gbps requires 256 matching units. Such a massive replication costs a lot of FPGA resources, which becomes the major bottleneck of the

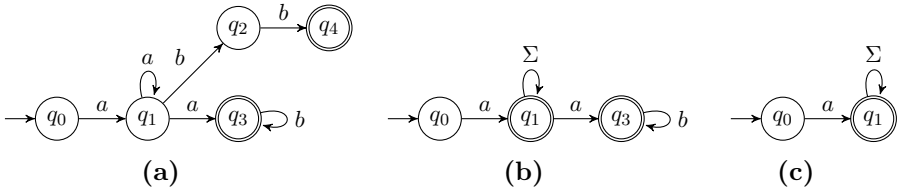


Figure 5.1: An example of the border-pruning reduction: the original NFA (a), after border-pruning (b), and after a subsequent simulation reduction (c).

approach. It is hence crucial to keep the NFAs as small as possible and for this reason, we propose lightweight approximate reductions of NFAs.

5.2 Samples Driven Approximate Reduction of NFAs

In this section, we propose two approaches for lightweight approximate reduction of NFAs. The *border-pruning reduction* is based on removing certain states from the NFA (and introducing self-loops), the *merging reduction* collapses similar parts of the NFA. Both reductions are steered by information about a typical traffic. In this section, we assume definitions from Chapter 2.

5.2.1 Border-pruning Reduction

Our first NFA reduction is the so-called *border-pruning reduction*. Do not confuse it with the pruning reduction in Chapter 4. Here we use pruning in a slightly different context. The reduction removes from the automaton a set R of states considered *insignificant*, together with all their adjacent transitions. At the same time, in order to overapproximate the original language, all states that are not removed and that have transitions going to a removed state are made final. Below, we call such states *border states*, forming a set B.

More precisely, let $\mathcal{A} = (Q, \delta, I, F)$ be an NFA over a fixed alphabet Σ, and let $R \subseteq Q$, such that $I \cap R \neq \emptyset$, be a set of states to be removed (we will later discuss how to find such a set). Let $B = \{q \in Q \setminus R \mid \exists a \in \Sigma : \delta(q, a) \cap R \neq \emptyset\}$ be the set of border states corresponding to R. The operation of border-pruning from $\mathcal{A}$ the states in R produces the NFA $\mathcal{A}_R = (Q' = Q \setminus R, \delta', I, F')$ where $\delta' = (\delta \cap (Q' \times \Sigma \times Q')) \cup \{q \xrightarrow{a} q \mid q \in B, a \in \Sigma\}$ and $F' = (F \cap Q') \cup B$.

The border-pruning reduction over-approximates the original language, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_R)$. The obtained NFA can, of course, be potentially further reduced by exact, simulation-based reductions [208].

The trade-off between reduction and accuracy that the border-pruning reduction offers depends on the choice of the set R of the states to be removed. We therefore try to compose R from such states of $\mathcal{A}$ that have the least influence on the acceptance/rejection of packets in typical traffic. For that, we use a representative sample $\mathcal{S}$ of the network traffic in the form of a multiset of packets. We label each state $q \in Q$ of the input NFA $\mathcal{A}$ by its *significance*: the number $\ell(q)$ of packets from $\mathcal{S}$ over which the state q can be reached in $\mathcal{A}$ (if there are multiple ways of reaching q over the same packet, we do not distinguish them). Formally, $\ell(q) = \sum\{\mathcal{S}(w) \mid w \in \{u.v \in \mathcal{S} \mid q_I \overset{u}{\rightsquigarrow} q, q_I \in I\}\}$ where $\mathcal{S}(w)$ is the number of occurrences of the packet w in the multiset $\mathcal{S}$.

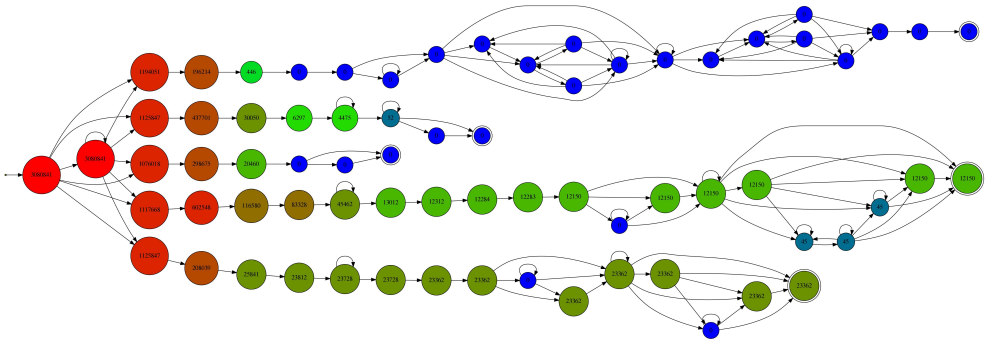


Figure 5.2: A heat map illustrating significance of states in a typical NFA.

The error caused by the border-pruning reduction based on a set of states R w.r.t. a sample $\mathcal{S}$ can be bounded in terms of significance of the border states B corresponding to R. Indeed, only the packets accepted at some border state can get wrongly accepted. Formally, $error_{ac}(\mathcal{S}, \mathcal{A}, \mathcal{A}_R) \leq \sum_{q \in B} \ell(q)$ where $error_{ac}(\mathcal{S}, \mathcal{A}, \mathcal{A}_R) = \sum\{\mathcal{S}(w) \mid w \in \mathcal{L}(\mathcal{A}_R) \setminus \mathcal{L}(\mathcal{A})\}$ is the exact error caused by the reduction on the sample $\mathcal{S}$.

To specify the desired reduction, we use a target *reduction ratio* $\theta \in \langle 0, 1\rangle$ meaning that the automaton should be reduced to $m = \lceil \theta \cdot |Q| \rceil$ states. To obtain m states while minimizing the error, we fill R with $|Q| - m$ least significant states[1]. The following example shows the pruning reduction of a simple NFA.

Example 5.2.1. *Consider an NFA from Figure 5.1a and let the reduction ratio be $\theta = 0.6$. Further assume that the significance of the states q_2 and q_4 is smaller than that of q_0, q_1, and q_3. Intuitively, this means that seeing b after the initial a symbols is rare. Hence, even if we allow the automaton to accept a string with the initial a symbols followed by a b symbol that is not followed by the second required b, no big error will be caused. Indeed the border-pruning reduction applied on the NFA chooses $R = \{q_2, q_4\}$ and hence $B = \{q_1\}$. The reduced automaton is shown in Figure 5.1b.*

The fact that scenarios, such as of Example 5.2.1, are common for NFAs obtained from real-world REs over real-world traffic is illustrated by our successful experiments (Section 5.3). As an additional illustration, Figure 5.2 shows a heat map of the states of an NFA obtained from one simplified RE (the `sprobe` PCRE mentioned in Section 5.3), having a typical structure of many of the NFAs that one obtains from real-world PCREs, which was labelled over a sample of real-world traffic. The "cold" states (blue, green) have low significance and are good candidates for border-pruning.

The significance of all states is computed efficiently in time $\mathcal{O}(kn^2)$ where $n = |Q|$ and $k = \sum_{w \in \mathcal{S}} |w| \cdot \mathcal{S}(w)$ is the overall length of $\mathcal{S}$. For that, we can use the subset construction known from determinization of NFAs, just run on particular packets $w \in \mathcal{S}$. Namely, each state's significance is initially set to zero. Then, we run $\mathcal{A}$ over every $w = a_1 \dots a_l \in \mathcal{S}$, computing consecutive sets of states Q^i that are possibly reached after processing the prefix $a_1 \dots a_i$, and, at the end of the run, we increment by one the significance of each of the states encountered on the way. Formally, for each $w \in \mathcal{S}$, we start with $Q^0 = I$, and,

[1] This strategy can cause a larger error when an originally non-accepting border state of a high significance is forced to become accepting by some insignificant accepting successor state. This could be avoided, e.g., by preferring border-pruning final states without a significant successor border state. In our experiments, we, however, sufficed with the simple strategy.

subsequently, compute $Q^{i+1} = \delta(Q_1^i, a_{i+1})$ for all $1 < i < l$. The significance of all states in the set $\bigcup_{i=0}^{l} Q^i$ is then incremented by one.

5.2.2 Merging Reduction

Our second reduction, called a *merging reduction*, is motivated by an observation that, in typical traffic, packets that start with a prefix of a certain kind (i.e., they are from some language L) almost always continue by an infix w that follows a predetermined pattern (a concrete word or a sequence of characters from predetermined character classes). We say that, in a sample $\mathcal{S}$, the pattern of w is *predetermined* by L. The part of the automaton that, after reading the prefix from L, tests whether the infix fits the pattern can be significantly simplified by collapsing it into a single state with a self-loop over all the symbols that label the original transitions while causing a small error only: Indeed, it is unlikely that a packet with an infix other than the predetermined one will appear after the given prefix. Note that the border-pruning reduction discussed previously is not suitable for simplifying the states that test the pattern as they may be of an arbitrarily high significance.

The operation of merging a sub-automaton based on a set S of states means to (i) redirect the targets of transitions entering S to a new state s, (ii) reconnect all transitions leaving S to start from s instead, (iii) make s final iff any of the states in S is final, and (iv) remove the states of S. Note that, like border-pruning, merging also over-approximates the language by allowing any permutation of the infix pattern.

Our detection of the parts of automata to be merged—typically, sequences of states—is based on a notion of *distance* defined for a pair of states q and r as $\mathit{dist}(q, r) = \max\left\{\frac{\ell(r)}{\ell(q)}, \frac{\ell(q)}{\ell(r)}\right\}$ if they are neighbors (i.e., $q \xrightarrow{a} r$ or $r \xrightarrow{a} q$ for some a) and as $\mathit{dist}(q, r) = \infty$ otherwise. Intuitively, a small $\mathit{dist}(q, r)$ means that $\ell(q)$ and $\ell(r)$ are similar, which typically happens if most of the packets reaching q continue to r or vice versa.[2] Symbols on transitions from q to r hence form a predetermined pattern of length 1. Therefore, merging q and r, and thus over-approximating the pattern, should cause a small error only. Merging of longer patterns is then achieved by merging multiple patterns of length 1.

The merging reduction is parameterized by a *distance ceiling* τ—we merge states with distance below τ. Formally, the sets of states to be merged are defined as the equivalence classes of the smallest equivalence $\sim_\tau \subseteq Q \times Q$ that contains all pairs (q, r) with $\mathit{dist}(q, r) \leq \tau$ (in other words, $\sim_\tau$ is the reflexive transitive closure of $\{(q, r) \mid \mathit{dist}(q, r) \leq \tau\}$).

The merging reduction does not provide theoretical guarantees, and it is to a large degree based on empirical experience, in which its parameterization with τ allows one to control the ratio between reduction and error well. There are, however, cases in which merging leads to an undesirable loss of precision even with a small τ. To limit such effects, we restrict merging by an additional parameter, the *frequency ceiling* $\gamma \in (0, 1]$. We prohibit merging of states with $\mathit{freq}(q) = \frac{\ell(q)}{|\mathcal{S}|} > \gamma$, that is, those whose frequency in $\mathcal{S}$ is larger than the ceiling. Formally, given τ and γ, we merge the equivalence classes of $\sim_\tau \cap \{(q, r) \mid \mathit{freq}(q) \leq \gamma \wedge \mathit{freq}(r) \leq \gamma\}$. The following example shows the merging reduction in action.

Example 5.2.2. *An example of the merging reduction is shown in Figure 5.3, assuming* $\mathit{dist}(q_2, q_3) < D$, $\mathit{dist}(q_3, q_4) < D$, *and that all other distances are greater than* D. *To make the states* q_2, q_3, *and* q_4 *less critical, we further assume that* $\mathit{freq}(q_2) < \gamma$, $\mathit{freq}(q_3) < \gamma$,

[2] In theory, it does not have to be the case, as $\mathit{dist}(q, r)$ may be polluted by packets reaching and leaving q and r from and to other states, but it is mostly the case in practice.

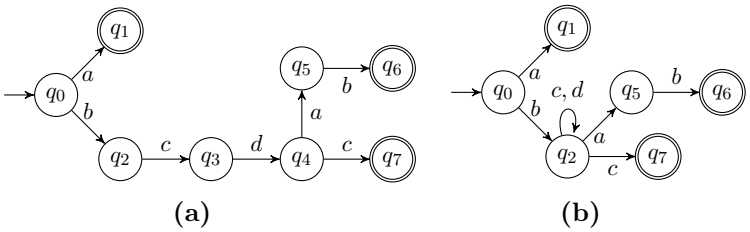


Figure 5.3: An input NFA (a) and an NFA obtained by merging (b).

Table 5.1: Sizes of the considered NFAs.

NFA	States	Transitions
`backdoor`	3,898	100,301
`l7-all`	7,280	2,647,620
`pop3`	923	209,467
`sprobe`	168	5,108
`spyware`	12,809	279,334

$freq(q_4) < \gamma$. Intuitively, this means that only some packets continue over b from q_0 to q_2. Roughly all of those packets then continue until they reach q_4, where another important split of the acceptance happens. The prefix b, however, predetermines the subsequent occurrence of cd. By simplifying the automaton and allowing any string from $\{c, d\}^*$ to appear after b, no significant error arises since most packets will anyway contain cd after b only.

5.3 Experimental Evaluation

In this section, we present an experimental evaluation of the proposed approach on real RE-matching instances.

Considered REs. We experimented with a set of REs describing protocols and attacks obtained from the L7 classifier for the Linux Netfilter [246] framework and from the Snort tool [269]. From the L7 classifier describing L7 protocols, we used all rules, giving us a set of REs denoted as `l7-all` below. From the Snort tool, we used the following set of REs: `backdoor`, `pop3`, and `spyware-put` (abbreviated as `spyware` below), describing attacks on selected protocols. We also used nine rules, denoted as `sprobe`, proposed for lawful interception in cooperation with our national police. We used the Netbench tool [234] for (i) translating REs to NFAs and (ii) the synthesis of reduced NFAs to VHDL. The sizes of the NFAs obtained by translating the considered REs are shown in Table 5.1.

Evaluation data. The sample of network traffic that we used for our experiments was obtained from two measuring points of a nation-wide Internet provider connected to a 100 Gbps backbone link. The training data used for labelling the automata contained ∼1M packets sampled from the captured traffic during the time of 19.5 min containing 509M packets. The testing data used for the subsequent evaluation consisted of ∼21M packets sampled from the captured traffic containing ∼210M packets. The testing data was sampled over the time of 105 hours (over 4 days) such that every hour 1M packets were captured.

Evaluation environment. We implemented the proposed techniques in a Python prototype[3]. In the experiments, we ran merging with the frequency ceiling $\gamma = 0.1$ and the distance ceiling $\tau = 1.005$. FPGA synthesis was done using Xilinx Vivado v.2018.1.

Running time. Our reduction techniques are light-weight, and, in contrast to the approaches presented in Chapter 4, they are capable of reducing very large NFAs appearing in real-world RE matching problems for real network scenarios. (Recall that the largest automata we consider have over 12k states or over 2.5M transitions.) Using the training data containing 1M packets, we needed about 15 min to derive the state labelling function ℓ for the largest considered NFAs. The runtime of the other parts of the reduction process was then negligible. Also note that, for a given NFA, the labelling can be performed only once for various values of the reduction ratio θ.

Research questions. We are interested in the following two key research questions related to the proposed approach:

(i) Are our reduction techniques able to provide useful trade-offs between the reduction error and the reduction ratio?

(ii) Can the reduced NFAs be compiled into an architecture with throughput of 100 Gbps and beyond?

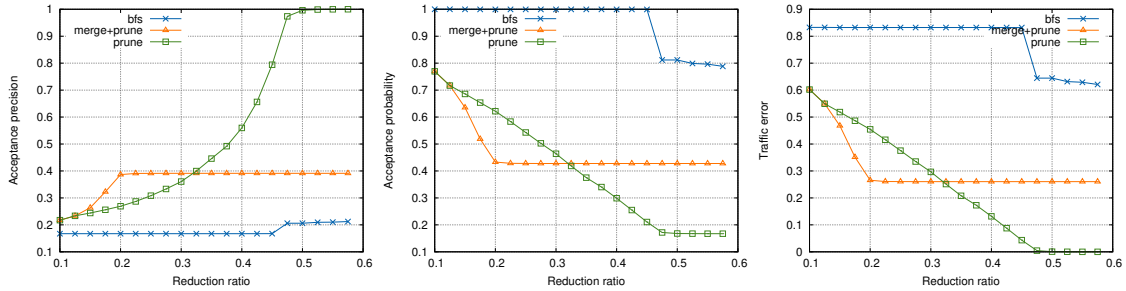
5.3.1 Reduction Trade-offs

In our experiments with the reduction techniques, we consider both the border-pruning and merging reductions. The merging reduction is, however, always combined with a subsequent border-pruning reduction as a standalone use of merging turned out not to be effective. Moreover, as a baseline, we also consider a so-called *bfs-reduction*. It does not use any training traffic and works by simply border-pruning of states that are far from the initial state. All of the approximate reductions are followed by the exact simulation-based reduction [206] whenever the tool REDUCE [208] implementing this reduction is capable of handling the approximate NFA.

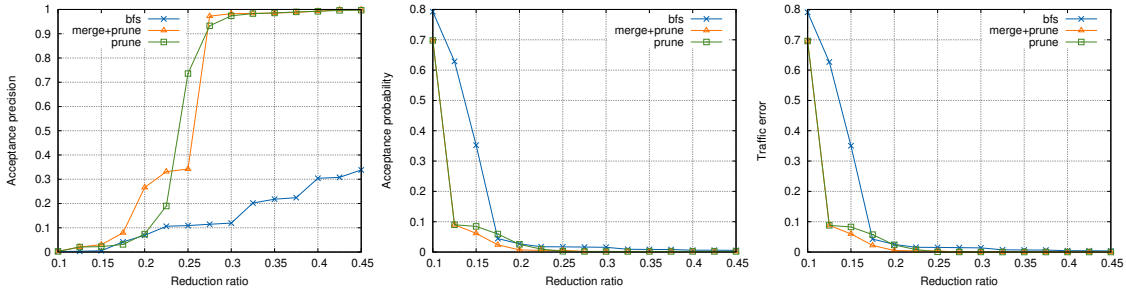
We consider two metrics characterizing the reduced automata. The first metric is the *acceptance precision* $AP = \frac{A_{TP}}{A_{FP}+A_{TP}}$ where A_{TP} denotes acceptance true positives (the packet is accepted and should have been accepted) and A_{FP} denotes acceptance false positives (the packet is accepted and should *not* have been accepted). Note that since our hardware architecture uses NFAs to accept packets based on their prefixes, we use adjusted condition of a string acceptance by a NFA in the experiments. A string w is accepted if arbitrary *prefix* of w is accepted according to the standard definition.

This metric expresses the ratio of correctly accepted packets to all accepted packets from the testing traffic sample and hence characterizes the error caused by the approximation. Our second metric is the *acceptance probability* $Prob = \frac{A_{TP}+A_{FP}}{|\mathcal{S}|}$ (where $|\mathcal{S}|$ denotes the size of the input network traffic sample) that captures what fraction of the input traffic is accepted by the reduced NFA and passed to the subsequent software NIDS, i.e., how much the NFA reduces the flow of packets to be further processed. Our last metric, similarly to Section 4.5, is the *traffic error* $TE = \frac{A_{FP}}{|\mathcal{S}|}$ capturing a ratio of misclassified packets from the testing sample.

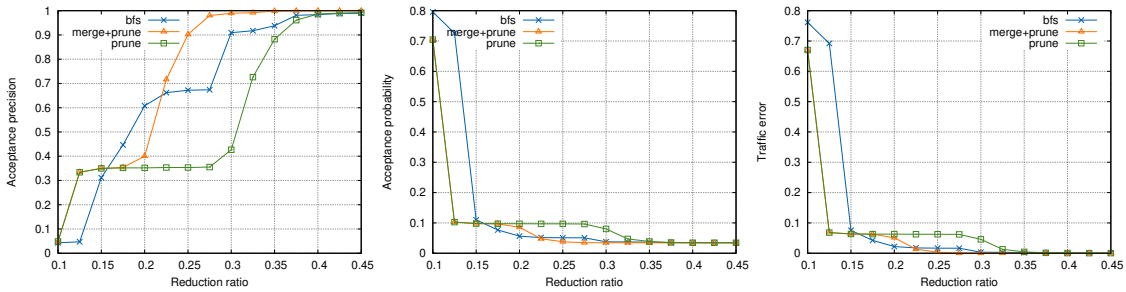
[3] `https://github.com/jsemric/ahofa`



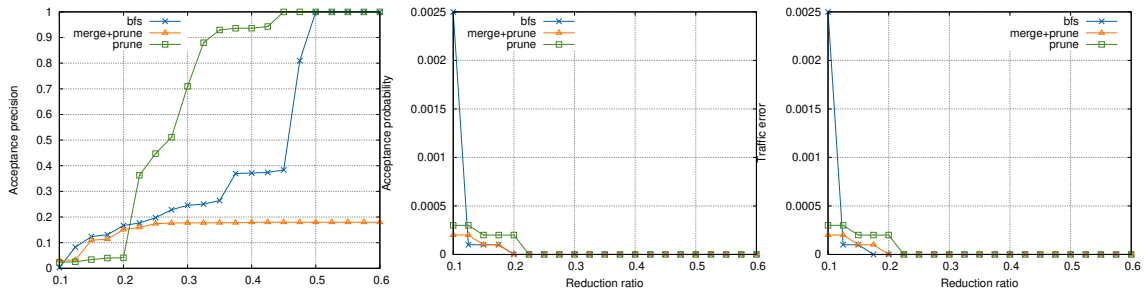
(a) From left: *AP*, *Prob*, and *TE* for `17-all`.



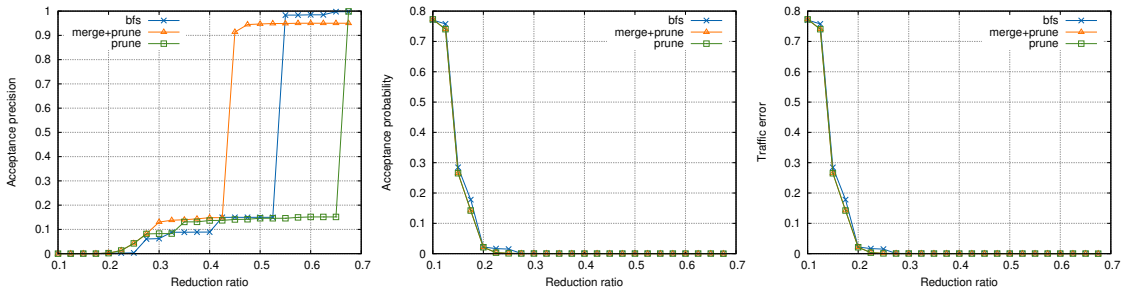
(b) From left: *AP*, *Prob*, and *TE* for `backdoor`.



(c) From left: *AP*, *Prob*, and *TE* for `spyware`.



(d) From left: *AP*, *Prob*, and *TE* for `pop3`.



(e) From left: *AP*, *Prob*, and *TE* for `sprobe`.

Figure 5.4: Results for `17-all`, `backdoor`, `spyware`, `pop3`, and `sprobe`.

Figures 5.4 show the trade-offs achieved by our different reduction strategies on challenging RE matching problems. In particular, *bfs*, *border-prune* (denoted as "prune"), and the combination of *merging* and *border-pruning* (denoted as "merge-prune").

Figure 5.4a shows results for the NFA describing `l7-all`. We observe that the particular reduction techniques provide a different quality of the trade-offs. In particular, *bfs* is not capable of producing any useful approximation. Further, we can observe that *merge-prune* dominates for reduction ratios lower than 0.3, but it is significantly outperformed by *prune* for higher ratios. The figures show that these trends are preserved for all our metrics. Note that, in this case, the original NFA accepts around 17 % of the traffic, and using the *prune* technique, we obtain a reduced NFA having only a half of the states with almost the same acceptance probability *Prob*.

The reduction trade-offs that we obtain for the NFA of the `backdoor` attack are plotted in Figure 5.4b. The *bfs* reduction is again significantly outperformed by both the *prune* and *merge-prune* methods when AP is considered. Note that these two techniques provide reductions that achieve almost a 100 % AP with negligible traffic error using only 35 % of the states of the original NFA. As the NFA accepts only 0.2 % of the traffic, we can obtain the accepting probability *Prob* that is close to this value using only 22 % of its states regardless of the reduction used.

We observed similar trends also for the smallest automata `pop3` and `sprobe` (see Figures 5.4d and 5.4e) where almost a 0 % *Prob* and *TE* was achieved using only 20 % and 25 % states, respectively. In the case of `sprobe`, AP for reduction ratios between 0.55 and 0.65 the *bfs* reduction outperforms both the *prune* and the *merge-prune* reductions.

Finally, we report the results for `spyware` REs in Figure 5.4c. We can see that, w.r.t. AP, *prune* lags behind the other two techniques for reduction ratios between 0.15 and 0.35 (similar trend as for `sprobe`). For higher reductions, all techniques provide AP close to 100 %. Similar trends can be observed also for the *Prob* metric. Note that the original NFA accepts about 3.5 % of the input traffic only.

The experiments conducted in this subsection clearly demonstrate that the proposed reduction techniques are able to provide high-quality trade-offs between the precision and the reduction factors. They also show that our techniques can outperform the baseline *bfs-reduction* and can handle very complex NFAs (having more than 10k states), where existing methods (such as those presented in Chapter 4) fail.

5.3.2 The Real Impact in an FPGA-Accelerated NIDS

We will use the reduced NFAs from the previous section to obtain instances of NFAs that can be used in FPGA-based NIDSes to effectively decrease the amount of traffic that the software part of the NIDS needs to process. We synthesize our designs for a card with the Xilinx Virtex UltraScale+ VU9P FPGA chip, which contains 1,182k LUTs (other resources are in our case always dominated by the number of LUTs). From our experience, it is possible to use up to 70 % of the LUTs available on the FPGA and successfully route designs at the considered frequency (200 MHz), which leaves us with 827k LUTs that we can use. Moreover, the components that we use for receiving packets and transferring them to the CPU consume around 90k LUTs, we are therefore left with 737k LUTs for RE matching. In the RE matching unit, we use the pipelined NFA architecture described in Section 5.1 instantiated with 8-bit data-width of the NFAs, which gave us the best results.

Our goal is to obtain single-box NIDSes using a combination of hardware preprocessing and a software NIDS. This means that the task of the hardware accelerator is to decrease

Table 5.2: FPGA resources of the architectures based on reduced NFAs. Settings exceeding FPGA resources are highlighted.

Speed	`backdoor`		`spyware`			`l7-all`	
	Precise	*2 % of tr.*	*Precise*	*4 % of tr.*	*5 % of tr.*	*Precise*	*43 % of tr.*
100	236k	149k	5M	227k	131k	1.8M	512k
200	473k	299k	10M	453k	262k	–	–
400	946k	597k	20M	907k	524k	–	–

the amount of the traffic entering the software part as much as possible while keeping all suspicious packets. Below, we provide results of our experiments for some of the considered sets of REs. We tried to compile architectures based on reduced NFAs for the speeds 100, 200, and 400 Gbps.

`backdoor`. In the first part of Table 5.2, we present consumption of FPGA resources for the Snort's `backdoor` module. The architecture with the original NFA (column *Precise*), can process traffic up to 200 Gbps only (the precise NFA consumes 3,695 LUTs). In order to process 400 Gbps, it is necessary to use a reduced NFA. The architecture based on the reduced NFA takes ∼597k LUTs. The reduced NFA decreased the amount of traffic sent to the CPU to 2 % (i.e., 8 Gbps). We stress that the reduced NFA has AP more than 93 % and TE around 0.01 % (Figure 5.4b) and therefore only a small fraction of packets are misclassified.

`spyware`. Our results for the Snort's `spyware` module are shown in the second part of Table 5.2. This module is much more complex than `backdoor`, since its precise NFA takes ∼78k LUTs (`spyware` has about 3 times more states than `backdoor`). Therefore, to obtain throughput of 100 Gbps and above, reduced NFAs are needed. For 100 Gbps and 200 Gbps we used a reduced NFA decreasing the amount of traffic sent to the CPU to 4 % (i.e., 4 Gbps for 100 Gbps and 8 Gbps for 200 Gbps). As in the previous case, the reduced NFA has a high precision (98 %) with the traffic error 0.04 % (Figure 5.4c). For 400 Gbps, more drastic reduction is needed. For this throughput we used a reduced NFA decreasing the amount of traffic sent to software below 5 % (i.e., 18.8 Gbps). Although 18.8 Gbps is on the edge of capabilities of current SW-based NIDSes we were still able to get quite high precision (∼72 %) with the traffic error 1.3 %.

`l7-all`. Our most challenging example is from the `l7-all` RE set. Although the size of the precise automaton is not as large as for `spyware` (the precise NFA for `l7-all` consumes 27,650 LUTs), it is less amenable for approximate reduction because, in contrast to Snort modules, it contains REs that are matched by many packets. The results for the `l7-all` RE set are shown in the third part of Table 5.2. Our best solution reduces the input traffic from 100 Gbps to 42.8 Gbps and uses ∼512k LUTs. The reduced NFA has a precision of 39 % with the traffic error 26 %.

`sprobe` and `pop3`. The sets of REs for `sprobe` and `pop3` are, on the other hand, quite less challenging. The precise NFAs consume only 195 and 1,721 LUTs, respectively, so we can easily obtain a precise design at 400 Gbps using ∼50k and ∼440k LUTs, respectively.

The experiments conducted in this section clearly demonstrate the practical potential of our approach. The key observation is that the resource reductions provided by the reduced NFAs directly depend on the characteristics of the underlying NFAs (both the precise NFA and the reduced variants) and the typical traffic. Apart from the size of the precise NFA, there are two crucial characteristics: (i) whether the number of packets accepted by the precise NFA is low and (ii) whether the reduction can compress the NFA while not increasing the number of accepted packets too much. If both these conditions are met (as for `backdoor` and `spyware`), we observe drastic resource savings allowing us to achieve throughput of the resulting NIDSes going beyond 100 Gbps, which is encouraging for REs of such size and complexity. On the other hand, if the original NFA is large, accepts many packets, and highly precise reductions achieve only moderate reductions (as for `l7-all`), our approach provides only moderate savings and ensuring 100 Gbps remains beyond the edge of what we can achieve.

5.4 Conclusion

In this chapter, we have leveraged techniques for lightweight approximate reduction of NFAs steered by a multiset of strings representing a typical network traffic. Our approximate reductions can handle large NFAs used in network traffic filtering. Moreover, these reductions allow RE matching of a set of Snort modules on speeds significantly beyond the capabilities of state-of-the-art single-box solutions, namely 100, 200, and even 400 Gbps, which proves a practical impact of our approach. The use of the approximate reduction allowed us to significantly decrease the size of the NFAs while keeping the number of false positives low (e.g., for Snort's `spyware` module, we obtained a reduction to 28 % of the original size while keeping the error below 2 %). The work on which this chapter is based was published in the proceedings of FCCM'19 [306] (the original paper [306] contains, moreover, a novel multi-stage architecture—not mentioned in this chapter—pushing the practical usability of this approach even further).

A possible direction of a further research can include a refinement of the border-pruning reduction with additional information about the significance of border states or a position of states in the automaton.

Part II:
Automata in Decision Procedures

Chapter 6

Automata in Decision Procedures

Mathematical logic, from its beginning in ancient Greece, is a strong tool for a precise description of facts, and forms a basis for exact reasoning. In the context of computer science, (formal) logic is often used to express properties of systems in a precise and concise way. For instance in formal verification, in particular model-checking, a formula in a suitable logic encodes the desired behavior of a system and it is then checked whether the system satisfies the formula.

Logic has attracted great interest of researchers from the beginning of 20th century when Hilbert formulated a program of axiomatization of mathematics to avoid inconsistencies and paradoxes. However, a few years later Gödel proved, in his famous incompleteness theorems, that this program is hard to achieve (basically by proving that even Peano arithmetic is incomplete and that it is not possible to prove consistency within the theory).

This result, together with the pioneering work of Turing and the blossom of computers launched a haunt for *decidable logics/theories* and their *efficient* decision procedures. Even though validity checking of first-order (FO) logic is undecidable, there are expressible theories/logics that are decidable, such as Presburger arithmetic [231], quantifier-free theory of strings [197], theory of real-closed ordered fields [266], or even various monadic second-order logics [66, 237]. A cricial requirement put on decision procedures is nowadays their efficiency, because the decision procedures form a fundamental stone in many applications ranging from software and/or hardware verification across synthesis of systems to artificial intelligence.

In this thesis, we, in particular, deal with automata in decision procedures. The connection between finite automata and logics was already proposed in seminal works [66, 65, 237, 97] relating *monadic second-order logics* (MSO) with finite automata. The decision procedures we propose, like the multiple classical decision procedures, use automata to represent (at least partially) all models of a given formula. However, note that the use of automata is not limited to represent models only, e.g., in Chapter 9, we use automata as an efficient representation of a proof graph in the context of the theory of strings. In this chapter, we focus on a brief introduction to automata-based decision procedures for the *weak monadic second-order logic of k successors* (WSkS) and for *Presburger arithmetic* as they are the basis for the following chapters.

Chapter outline. This chapter serves as an introduction to WSkS and Presburger arithmetic for the following chapters. Section 6.1 introduces necessary definitions related to finite tree and word automata. Section 6.2 focuses on the definition of syntax and semantics of WSkS, its decision procedures (as presented in [138, 141]), and applications of WSkS.

Section 6.3 then deals with Presburger arithmetic, its definition, decision procedures, and applications. Finally, Section 6.4 touches upon the complexity results, expressivity, and the SkS logic.

6.1 Preliminaries

In this chapter, we assume the definitions related to functions, trees, tree automata, and word automata from Chapter 2. We extend them with definitions used in the rest of the chapter.

Tree automata. Let $\mathcal{A} = (Q, \Sigma, \delta, I, R)$ be a TA. We extend δ to a set of symbols $\Gamma \subseteq \Sigma$ as $\delta_\Gamma = \bigcup\{\delta_a(q_1, \ldots, q_k) \mid a \in \Gamma\}$. Let Σ' be an alphabet and $\pi : \Sigma' \to 2^\Sigma$ be a function. Then the *projection* of $\mathcal{A}$ w.r.t. π is defined as $\pi(\mathcal{A}) = (Q, \Sigma', \delta', I, R)$ where $\delta'_a(q_1, \ldots, q_k) = \delta_{\pi(a)}(q_1, \ldots, q_k)$. The set $\mathit{reach}_\delta(S)$ of states *reachable* from a set $S \subseteq Q$ through δ-transitions is computed as the least fixpoint

$$\mathit{reach}_\delta(S) = \mu Z.\ S \cup \bigcup_{q_1,\ldots,q_k \in Z} \delta(q_1, \ldots, q_k). \tag{6.1}$$

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

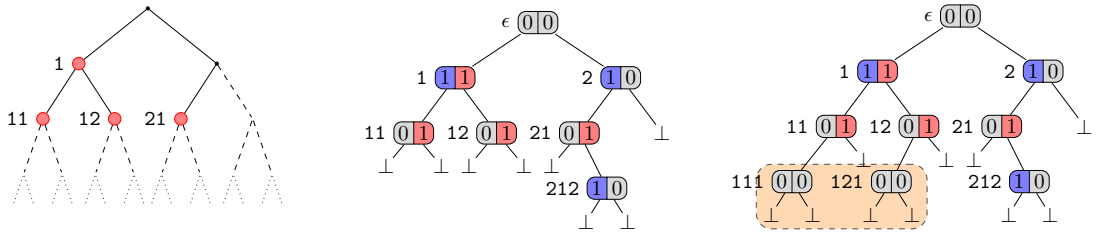
Word automata. Although word automata can be seen as 1-ary tree automata where unary trees are read in the top-down manner (not bottom-up), for the sake of simplicity we use the formalism of NFAs in the decision procedure of Presburger arithmetic. For this reason, we adjust the operations defined for TAs in the previous paragraph also to NFAs. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be an NFA, Σ' be an alphabet, and $\pi : \Sigma' \to 2^\Sigma$ be a function. Then the *projection* of $\mathcal{A}$ w.r.t. π is defined as $\pi(\mathcal{A}) = (Q, \Sigma', \delta', I, F)$ where $\delta'(q, a) = \bigcup_{b \in \pi(a)} \delta(q, b)$. Similarly to the case of TAs, the *derivative* of $\mathcal{A}$ with respect to a^* for some $a \in \Sigma$ is defined as $\mathcal{A} - a^* = (Q, \Sigma, \delta, I, \mathit{reach}_{\delta^{-1}(\cdot,a)}(F))$.

6.2 Weak Monadic Second-order Logic of k Successors

Weak monadic second-order logic of k successors allows reasoning about regular properties of k-ary trees. As the name suggests it is a logic that allows quantification over second-order *variables*, which we denote by upper-case letters $X, Y, \ldots$ and range over *finite sets* of tree positions in $\{\mathtt{1}, \ldots, \mathtt{k}\}^*$. The *weak* in the name reflects finiteness of variable assignments and k denotes the number of successors. In this section, we describe syntax and semantics of WSkS, its decision procedures, and in the last part also applications of WSkS. This section is based on definitions and notations used in [138, 141].

6.2.1 Syntax and Semantics

Atomic formulae (atoms) of WSkS are of the form (i) $X \subseteq Y$ and (ii) $X = S_\mathtt{i}(Y)$ for $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{k}\}$. Formulae are constructed from atoms using the logical connectives $\wedge, \neg$, 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 $\mathrm{Sing}(X)$



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

Figure 6.1: Consider a set of variables $\mathbb{X} = \{X, Y\}$ and an assignment $\kappa = \{X \mapsto \{1, 2, 212\}, Y \mapsto \{1, 11, 12, 21\}\}$ in WS2S. In (a), we show positions assigned to variable Y. The minimal encoding of κ into a binary tree is shown in (b), and an encoding that is not minimal is shown in (c).

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 $\neg(X = \emptyset) \wedge \forall Y.\ Y \subseteq X \rightarrow ((Y \subseteq X \wedge X \subseteq Y) \vee Y = \emptyset)$). Despite it is not a basic connective, we use the disjunction in optimizations presented in the following chapters, so we mention it below too.

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

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

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

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

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

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

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

Informally, the $S_i(Y)$ function returns all positions from Y shifted to their i-th child. Satisfaction of formulae built using Boolean connectives and the quantifier is defined as usual. A formula φ is *valid*, written $\models \varphi$, iff all assignments of its free variables are its models, and *satisfiable* iff it has a model. W.l.o.g., we assume that each variable in a formula either has only free occurrences or is quantified exactly once.

6.2.2 Representing Models as Trees

Let $\mathbb{X}$ be a finite set of variables. A *symbol* ξ over $\mathbb{X}$ is a (total) function $\xi : \mathbb{X} \rightarrow \{0, 1\}$; e.g., $\xi = \{X \mapsto 0, Y \mapsto 1\}$ is a symbol over $\mathbb{X} = \{X, Y\}$. We use $\Sigma_{\mathbb{X}}$ to denote the set of all symbols over $\mathbb{X}$ and $\vec{0}$ to denote the symbol mapping all variables in $\mathbb{X}$ to 0, i.e., $\vec{0} = \{X \mapsto 0 \mid X \in \mathbb{X}\}$.

A finite assignment $\nu : \mathbb{X} \rightarrow 2^{\{1,\ldots,k\}^*}$ of the free variables of a formula φ can be encoded as a finite tree τ_ν of symbols over $\mathbb{X}$ where every position $p \in \{1, \ldots, k\}^*$ satisfies the following conditions: (a) if $p \in \nu(X)$, then $\tau_\nu(p)$ contains $\{X \mapsto 1\}$, and (b) if $p \notin \nu(X)$,

then either $\tau_\nu(p)$ contains $\{X \mapsto 0\}$ or $\tau_\nu(p') = \bot$ for some prefix p' of p (note that the occurrences of $\bot$ in τ are limited since τ still needs to be a tree). Observe that ν can have multiple encodings: the unique minimal encoding τ_ν^{min} and (infinitely many) extensions of τ_ν^{min} with $\vec{0}$-only trees. See Figure 6.1 for an example of an assignment and its encodings. The *language* of φ is defined as the set of all encodings of its models $\mathcal{L}(\varphi) = \{\tau_\nu \in \Sigma_{\mathbb{X}}^\star \mid \nu \models \varphi \text{ and } \tau_\nu \text{ is an encoding of } \nu\}$.

Let ξ be a symbol over $\mathbb{X}$. For a set of variables $\mathbb{Y} \subseteq \mathbb{X}$, we define the *projection* of ξ 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 $\vec{0}$ with respect to $\{X\}$ is given as $\pi_X(\vec{0}) = \{\{X \mapsto 0, Y \mapsto 0\}, \{X \mapsto 1, Y \mapsto 0\}\}$. Further, we define the *inverse projection* of ξ with respect to $\mathbb{Y}$ as $\pi_{\mathbb{Y}}^\flat(\xi) = \{\xi_{|\mathbb{Y}}\}$. Intuitively, the inverse projection keeps only those assignments of variables that belong to $\mathbb{Y}$. Consider for example the symbol $\xi = \{X \mapsto 1, Y \mapsto 0, Z \mapsto 1\}$ over $\mathbb{X} = \{X, Y, Z\}$. The inverse projection of ξ w.r.t. $\mathbb{Y} = \{Y, Z\}$ is given as $\pi_{\mathbb{Y}}^\flat(\xi) = \{\{Y \mapsto 0, Z \mapsto 1\}\}$. 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 $\{\tau' \in \Sigma_{\mathbb{X}}^\star \mid \forall p \in \mathrm{dom}(\tau) :$ if $\tau(p) = \bot$, then $\tau'(p) = \bot$, else $\tau'(p) \in \pi_{\mathbb{Y}}(\tau(p))\}$ and subsequently to languages L so that $\pi_{\mathbb{Y}}(L) = \bigcup\{\pi_{\mathbb{Y}}(\tau) \mid \tau \in L\}$. All these definitions can be naturally extended also to the inverse projection.

6.2.3 Decision Procedure for WSkS

In this section, we focus on decision procedures for WSkS. The first part is devoted to the decision procedure based on the (tree) automata-logic connection as introduced in [66, 268, 98]. The second part then deals with various approaches to deciding WSkS and optimizations of the classical decision procedure as well as with decision procedures of closely related logics.

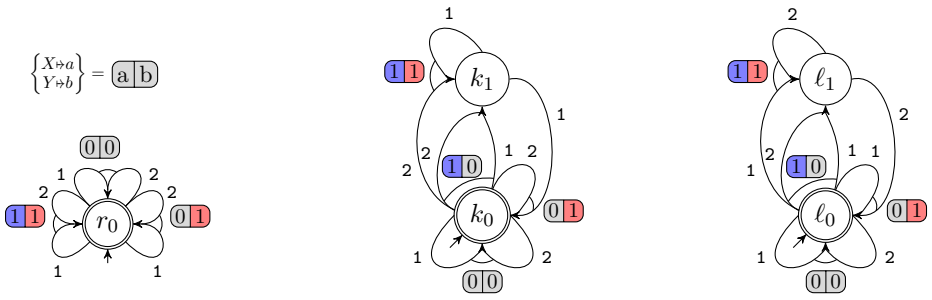
The classical decision procedure for WSkS. The classical decision procedure for the WSkS logic goes through a direct construction of a TA $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, R)$ having the same language as a given formula φ (see [82] for more detailed description). The satisfiability checking of φ is then equivalent to the emptiness test of $\mathcal{A}_\varphi$, which can be implemented through the equivalence $\mathcal{L}(\mathcal{A}_\varphi) \neq \emptyset$ if and only if $reach_\delta(I) \cap R \neq \emptyset$.

The automaton $\mathcal{A}_\varphi$ is constructed by induction to the structure of φ. Namely, if φ is an atomic formula with free variables $\mathbb{X}$, then $\mathcal{A}_\varphi$ is a pre-defined *base* TA over $\Sigma_{\mathbb{X}}$ (we show those TAs for WS2S in Figure 6.2). Otherwise, if φ is not atomic, then $\mathcal{A}_\varphi$ is built from the automata corresponding to subformulae of φ as follows.

(i) If $\varphi = \psi_1 \wedge \psi_2$ then, for $\mathcal{A}_{\psi_1}$ over alphabet $\Sigma_{\mathbb{X}}$ and $\mathcal{A}_{\psi_2}$ over alphabet $\Sigma_{\mathbb{Y}}$ the $\mathcal{A}_\varphi$ over alphabet $\Sigma_{\mathbb{X}\cup\mathbb{Y}}$ is given as $\mathcal{A}_\varphi = \pi_{\mathbb{X}}^\flat(\mathcal{A}_{\psi_1}) \cap \pi_{\mathbb{Y}}^\flat(\mathcal{A}_{\psi_2})$. The $\pi^\flat$-transformation of automata $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$, respectively, into automata with compatible alphabets over variables $\mathbb{X} \cup \mathbb{Y}$ is called the *cylindrification*.

(ii) If $\varphi = \psi_1 \vee \psi_2$ then, for $\mathcal{A}_{\psi_1}$ over alphabet $\Sigma_{\mathbb{X}}$ and $\mathcal{A}_{\psi_2}$ over alphabet $\Sigma_{\mathbb{Y}}$, analogically to the previous case, $\mathcal{A}_\varphi = \pi_{\mathbb{X}}^\flat(\mathcal{A}_{\psi_1}) \cup \pi_{\mathbb{Y}}^\flat(\mathcal{A}_{\psi_2})$.

(iii) If $\varphi = \neg\psi$, then $\mathcal{A}_\varphi = \mathcal{A}_\psi^{\complement}$.



(a) $\mathcal{A}_{X \subseteq Y}$ **(b)** $\mathcal{A}_{X=S_1(Y)}$ **(c)** $\mathcal{A}_{X=S_2(Y)}$

Figure 6.2: Tree automata for atomic WS2S formulae. Transitions are represented using multiple-source hyper-edges. For instance, the transition $(k_0, k_1) \xrightarrow{1\,1} k_1$ in $\mathcal{A}_{X=S_1(Y)}$ is represented by the hyper-edge with sources k_0 and k_1 over the symbol $1\,1 = \{X \mapsto 1, Y \mapsto 1\}$ that joins just before entering k_1. The 1 and 2 labels on the "legs" of the hyper-edge going to k_0 and k_1 denote the position in the left-hand side of the transition (1 and 2 stand for "first" and "second").

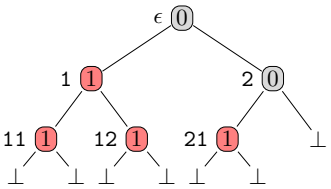


Figure 6.3: The minimal encoding of a model after the projection

(iv) If $\varphi = \exists X.\, \psi$, then the automaton $\mathcal{A}_\varphi$ is constructed as $det(\pi_X(\mathcal{A}_\psi)) - \vec{0}^\star$. The projection implements the quantification by forgetting the values of the X component of all symbols. Since this yields non-determinism, projection is followed by determinization by the subset construction and saturation of leaf states.

Points (i)–(iii) are self-explanatory. In the case of point (iv), saturating the set of leaf states is needed to ensure that *every* encoding of every model is accepted. Indeed, if some were not accepted, the inductive construction could produce a wrong result. For instance, language complementation would not complement the set of encoded models since some of the encoded models could possibly belong to the original language and its complement.

Consider, for example, a formula ψ having the language $\mathcal{L}(\psi)$ given by the tree τ_ν in Figure 6.1b and all its $\vec{0}$-extensions. To obtain $\mathcal{L}(\exists X.\psi)$, it is not sufficient to forget the values of the X component because the projected language does not contain the minimal encoding shown in Figure 6.3, only its extensions (the minimal one would hence be present in the complement of the language). The saturation of the set of the leaf states includes the minimal encoding into the language. See [82] for more details.

Related decision procedures and optimizations. The classical automata-based procedure for WS1S/WS2S is implemented in the MONA tool [143, 105]. MONA uses deterministic TAs, represented by multiterminal BDDs, with a procedure based on eager minimization [173, 172]. On top of that MONA employs various heuristics, e.g., automata with "don't care" states or optimized handling of first-order variables [171]. Observations and

heuristics related to the implementation of WS1S decision procedure were also summarized in [125].

Although MONA achieves a good overall performance, there are still formulae that are too difficult for MONA, such as formulae with quantifier alternations requiring series of determinization and complementation steps. The works [111, 112] overcome this issue in the context of WS1S by a symbolic representation of NFAs with an on-demand evaluation of the expensive automata operations. A similar decision procedure of WS1S based on an idea of Brzozowski's derivative was studied in [273].

Regarding the logics closely related to WSkS the *monadic second-order logic of strings* (MSO(STR)) can be implemented using a similar automaton-based decision procedure (e.g., MONA implements MSO(STR) on the top of the WS1S decision procedure). The classical automata-based decision procedure of MSO(STR) is implemented (except of MONA), e.g., within the JMOSEL tool [272]. In particular, JMOSEL uses second-order value numbering reducing the amount of redundancy during the automata construction [199]. Another approach to the MSO(STR) decision procedure employs symbolic finite automata (automata with transitions labelled with predicates over some effective Boolean algebra) [90].

As we mentioned at the beginning, WSkS can express regular properties of trees. The work [121] brings a decision procedure, based on Shelah's composition method, for a MSO over inductive structures, allowing reasoning about more general structures (structures with bounded clique width).

The mentioned decision procedures for WSkS were more model-based approaches. In the end, we mention a complete axiomatization of MSO over finite trees [123] that could open doors to decision procedures of WSkS based on automated theorem proving.

6.2.4 Applications

The WSkS logic, and in particular the solver MONA, found a various range of applications. The first application field covers the *verification of pointer programs* with dynamic liked data structures. In [157] the authors proposed the verification of restricted Pascal-based pointer programs using MSO(STR). Later on, the verification of pointer programs with complex data structures using encoding the partial specifications into WSkS was considered [215]. Verification of pointer programs using separation logic extended with arithmetic and set constraints was studied in [78]. Combination of constraints allows to express reachability properties (e.g., capturing all nodes of a list to prove preservation of elements during sorting). The arithmetic and set constraints are expressed in Presburger arithmetic/WS1S and solved by MONA. The verification based on separation logic is not the only considered direction. In [195, 196] the authors present the Strand logic allowing to express a combination of properties of heaps with properties of the heap nodes. Strand is then used to reason about the correctness of pointer programs. The decision procedure is based on a translation into WSkS. A slightly different approach was proposed in [302] where the verification of linked data structures is performed by verifying specifications written in *higher-order* logic (HOL). Formulas in HOL are splitted and approximated to obtain formulas of decidable fragments of various logics including WSkS.

Applications of WSkS are not limited only to verification of pointer programs but include also *verification of string transformation programs* via encoding the programs in MSO(STR) [267] or verification of programs with arrays [305]. In the later, the authors propose an expressive FO theory of arrays of bounded elements with a decision procedure

based on a reduction to WS1S. Other related topics include verification of protocols [256] or reactive systems [174, 170].

To conclude the applications in verification, we give a brief overview related to *verification of parametric systems* and *hardware verification*. The work [33] focuses on verification of parametric networks of finite-state processes. Their approach is based on a representation of networks by a WS1S transition system (system with transitions described in WS1S), which is further abstracted into a finite state system and then analyzed by model-checking techniques. Another approach was proposed in [58] where the authors propose a technique for the verification of safety properties of parametric systems using the automata-logic connection of WS1S. A tool support for the verification of parametric was proposed in [48] where the authors bring a high level interface for MONA used to express parametric systems, their abstraction, and the validation of safety properties. Regarding hardware verification we mention verification of sequential circuits based on their description in MSO(STR) [32].

Finally, applications of WSkS in synthesis include *synthesis of sequential systems* [22], synthesis of safety controllers for web services [249], synthesis of a control program based on specification in MSO(STR) [152], or synthesis of functions from regular specifications [135]. As a concluding remark we mention an application of WSkS in linguistics [217], or the work [155] dealing with the satisfiability checking of a fragment of separation logic using translation of a formula into MSO over structures with bounded tree-width.

6.3 Presburger Arithmetic

Presburger arithmetic is a first-order theory allowing reasoning about linear expressions over natural numbers. Presburger arithmetic contains the *addition* operation with the relation of comparison $\leq$ (in the literature Presburger arithmetic used to be defined with equality instead of $\leq$ but for the purposes of the automata based approach we use $\leq$). Note that multiplication is not included as it leads to the undecidable Peano arithmetic. In this section, we give a brief description of syntax and semantics of Presburger arithmetic, its decision procedures, and in the last part we list also some of the applications.

6.3.1 Syntax and Semantics

Regarding the syntax, we start with the definition of terms. Terms are constructed using the following grammar:

$$t ::= x \mid 0 \mid 1 \mid t + t, \tag{6.2}$$

where x is a first-order variable, which we denote by lower-case letters, ranging over numbers from ω. Atomic formulae (atoms) are built from terms using the only predicate symbol $\leq$. Formulae are constructed from atoms using the connectives $\vee$, $\neg$, and the quantifier $\exists x$. Other terms (e.g., $n = \underbrace{1 + \cdots + 1}_{n-\text{times}}$ or $nx = \underbrace{x + \cdots + x}_{n-\text{times}}$ where x is a variable and $n \in \omega$), predicates (e.g., $<$, $=$), and connectives (e.g., $\wedge, \forall$) can be obtained as syntactic sugar.

A model of a Presburger formula $\varphi(\mathbb{X})$ with free variables $\mathbb{X}$ is a mapping $\sigma : \mathbb{X} \to \omega$ assigning natural numbers to free variables of φ. Satisfaction of a formula φ under σ, denoted as $\sigma \models \varphi$, is defined as follows:

(i) $\sigma \models t_1 \leq t_2$ iff $[t_1]_\sigma \leq [t_2]_\sigma$,

(ii) $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$,

(iii) $\sigma \models \neg\varphi$ iff not $\sigma \models \varphi$, and

(iv) $\sigma \models \exists x.\ \varphi$ iff there is $a \in \omega$ s.t. $\sigma \triangleleft \{x \mapsto a\} \models \varphi$,

and the valuation $[\cdot]_\sigma$ of a term t w.r.t. σ is inductively defined as (i) $[0]_\sigma = 0$, $[1]_\sigma = 1$, (ii) $[x]_\sigma = \sigma(x)$, and (iii) $[t_1 + t_2]_\sigma = [t_1]_\sigma + [t_2]_\sigma$. *Validity* and *satisfiability* of a formula φ are defined as usual.

6.3.2 Representing Models as Words

In this section we, using the notions of Section 6.2.2, encode models of a Presburger formula $\varphi(\mathbb{X})$ as finite words of symbols over $\mathbb{X}$. For that, we first define the function $bin : \omega \to \{0,1\}^*$ s.t. for a number $n \in \omega$, $bin(n)$ is the least-significant-bit-first *binary encoding* of n. For example, $bin(12) = 0011$. An assignment $\sigma : \mathbb{X} \to \omega$ of the free variables of formula φ is then encoded by a finite word w_σ over $\Sigma_\mathbb{X}$ as follows: (a) if $bin(\sigma(x))_i = v$ then w_σ contains $\{x \mapsto v\}$ at position i, and (b) if $|w_\sigma| > |bin(\sigma(x))|$ then w_σ contains $\{x \mapsto 0\}$ at all positions $|bin(\sigma(x))| \leq i < |w_\sigma|$. Intuitively, w_σ encodes the value of $\sigma(x)$ for each $x \in \mathbb{X}$ as a binary number. As in the case of WSkS, an assignment σ can be encoded using multiple words; the unique *minimal* encoding w_σ^{min} and encodings extending w_σ^{min} with a suffix from $\vec{0}^*$. The language of φ is then defined as a set of words encoding models of φ. In particular, $\mathcal{L}(\varphi) = \{w_\sigma \in \Sigma_\mathbb{X} \mid \sigma \models \varphi \text{ and } w_\sigma \text{ is an encoding of } \sigma\}$. For example, consider a formula $\psi \triangleq x = 2 \wedge y = 0$. The language of ψ is given as $\mathcal{L}(\psi) = \boxed{0}\boxed{0} \,.\, \boxed{1}\boxed{0}.\{\boxed{0}\boxed{0}\}^*$ where $\boxed{a}\boxed{b}$ denotes the symbol $\{x \mapsto a, y \mapsto b\}$.

6.3.3 Decision Procedure for Presburger Arithmetic

This section is devoted to decision procedures for Presburger arithmetic. In the first part, we describe the automata-based procedure based on translation of a formula into an automaton having the same language. The automata-based decision procedure is based on an idea of [65] and was further studied, e.g., in [57, 290]. In the second part, we give a brief overview of other decision procedures for Presburger arithmetic.

The automata-based decision procedure. The automata-based decision procedure constructs (as in the case of WSkS) for a Presburger formula φ the NFA $\mathcal{A}_\varphi$ s.t. $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$. Satisfiability checking of φ is implemented through checking emptiness of $\mathcal{A}_\varphi$. The emptiness checking for NFAs is realized via checking of reachability of a final state. The automaton $\mathcal{A}_\varphi$ is then constructed inductively w.r.t. the structure of φ as follows:

(i) For an atomic formula of the form $\varphi = \boldsymbol{a} \cdot \boldsymbol{x} \leq b$ with free variables $\mathbb{X}$ where $\boldsymbol{a} \in \mathbb{Z}^n$, $b \in \mathbb{Z}$, and $\boldsymbol{x} \in \mathbb{X}^n$, the automaton $\mathcal{A}_\varphi$ is constructed iteratively in a way that the languages of its states $q \in \mathbb{Z}$ encode all $\boldsymbol{\beta} \in \omega^n$ s.t. $\boldsymbol{a} \cdot \boldsymbol{\beta} \leq q$. See [106] for more details. An example of an atomic formula with the corresponding NFA is shown in Figure 6.4.

(ii) If $\varphi = \psi_1 \vee \psi_2$ then, for $\mathcal{A}_{\psi_1}$ over alphabet $\Sigma_\mathbb{X}$ and $\mathcal{A}_{\psi_2}$ over alphabet $\Sigma_\mathbb{Y}$, the automaton $\mathcal{A}_\varphi$ is constructed as $\mathcal{A}_\varphi = \pi^\flat_\mathbb{X}(\mathcal{A}_{\psi_1}) \cup \pi^\flat_\mathbb{Y}(\mathcal{A}_{\psi_2})$.

(iii) If $\varphi = \neg\psi$, then $\mathcal{A}_\varphi = \mathcal{A}^\complement_\psi$.

(iv) If $\varphi = \exists x.\ \psi$, then the automaton $\mathcal{A}_\varphi$ is constructed as $det(\pi_x(\mathcal{A}_\psi)) - \vec{0}^*$.

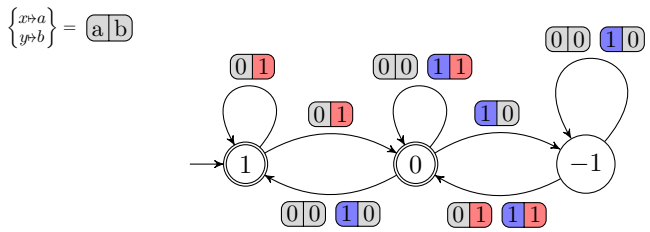


Figure 6.4: NFA $\mathcal{A}_\psi$ for the atomic formula $\psi \triangleq x - 2y \leq 1$.

The points (ii)–(iv) directly corresponds to the WSkS decision procedure (with a different kind of automata though). The main difference is in the construction of the NFAs for atomic formulae.

Other decision procedures for presburger arithmetic. The automata-based decision procedure of Presburger arithmetic is implemented, e.g., within the tools Lash [49] or Mona. The automata-based approach is not the only way to decide Presburger arithmetic. The original argument of decidability of Presburger arithmetic, given by Presburger himself, was based on *quantifier elimination* [231]. A variant of quantifier elimination was further given by Cooper [84]. Note that Presburger arithmetic itself does not support quantifier elimination–the theory needs to be augmented with the divisibility predicate.

Except of the aforementioned approaches for solving full Presburger arithmetic (i.e., including quantifiers), there are techniques aiming at fragments of Presburger arithmetic, in particular the quantifier-free fragment. In this case, the satisfiability of an atomic formula can be checked using the Branch-and-bound method (or its variations, for instance [103]) and a conjunction of atoms can be checked using the Omega test (see e.g., [178] for more details). Some of these techniques are implemented within state-of-the-art SMT solvers, such as Z3 [218] or CVC4 [31].

6.3.4 Applications

Presburger arithmetic found its applications in various fields of computer science. In this section, we pick up just a few results relating applications of Presburger arithmetic in formal verification, in particular in verification of programs with integer variables.

Linear relation analysis (LRA) [86], as an application of abstract interpretation, represents control points of a program by a system of linear inequalities expressing possible values of variables at that point in order to approximate the reachable configurations. In some cases, it is possible to give even a precise representation of reachable configurations (such a technique is called *acceleration*). The goal of this approach is to represent a transition closure of program loops (in other words the set of reachable configurations) in a precise way, e.g., using a Presburger formula [51, 83, 59, 30]. A combination of acceleration with different approaches was further used to refine verification of integer programs. In particular, model-checking based on predicate abstraction with interpolation combined with acceleration [70, 147], or abstract acceleration in LRA (if it is possible compute the precise closure using acceleration, otherwise use widening) [128, 129].

Slightly different approaches to verification using Presburger arithmetic include verification of concurrent systems with integer variables using a symbolic encoding of transition systems by Presburger formulae [67], or interpolation procedure for quantifier-free

Presburger arithmetic with uninterpreted predicates for the verification of programs with arrays [61].

Outside the field of verification, we mention the application in data array dependence analysis, where Presburger arithmetic is used to describe the flow of values [232]. The tool for dependence analysis is included in the Omega library containing also support for manipulating Presburger formulae [233].

6.4 Complexity, SkS, and Expressivity

In Section 6.2, we defined the WSkS logic in the way to be suitable for the underlying automata decision procedure. However, from a more logic-theoretic point of view, the WSkS logic can be also seen as weak monadic second-order logic over the structure of $(\{\mathtt{1},\ldots,\mathtt{k}\}^*,.\mathtt{1},\ldots,.\mathtt{k})$ where the function symbol $.\mathtt{i}$ for $\mathtt{1} \leq \mathtt{i} \leq \mathtt{k}$ is defined as the concatenation of the symbol $\mathtt{i}$ at the end of a string. More specifically if we consider unary encoding of natural numbers, WS1S can be seen as the weak monadic second-order logic over the structure $(\omega, +1)$. Similarly, Presburger arithmetic, defined in Section 6.3, can be seen as the first-order logic over the structure $(\omega, +)$.

Complexity. Intuitively, the automata-based decision procedure for WSkS can suffer from series of exponential blowups caused by determinization of automata. Indeed, in [264] a **NONELEMENTARY** upper bound already for WS1S was given. If we take into account the automata-based decision procedure, there is a **NONELEMENTARY** lower bound for a translation of WS1S formulae to automata. On top of that, the theory of WS1S is complete for a **NONELEMENTARY** complexity class meaning that there is no principally better method for deciding WS1S [242].

In the case of Presburger arithmetic, the situation is slightly more intricate. In [113] **2-NEXP** lower bound and in [41] **2-EXPSPACE** upper bound for Presburger arithmetic was given. Regarding the decision procedures, the worst-case complexity of Cooper's quantifier elimination was proven to be **3-EXP** [221]. Although the automata-based decision procedure is similar to the WSkS decision procedure, which has a **NONELEMENTARY** complexity, the complexity is not the case for Presburger arithmetic. As a matter of fact, the number of states of the minimal DFA for a formula is at most triple exponential in the size of the formula [169]. Moreover in [100] the authors proved the **3-EXP** worst-case complexity of an automata-based decision procedure based on an analysis of the structure of the automata obtained during the construction.

The SkS logic. The assumption of quantification over finite sets is sufficient for many applications. If we relax this assumption we obtain SkS (monadic second-order logic with k successors). The classical decision procedure for SkS is based, as in the case of WSkS, on the direct translation of formulae into automata [65, 237]. Models of an SkS formula are encoded as infinite k-ary trees (the encoding described in Section 6.2.2 can be naturally generalized also to infinite trees). For this reason, an automaton model accepting infinite trees is used. The background automaton model for SkS is usually *Muller tree (top-down) automaton*[1]. An infinite tree is accepted by this type of automaton if on each infinite path of the corresponding infinite run tree, the Muller accepting condition is met (see Section 10.1

[1]Rabin and Strett tree automata can be used as well. Büchi tree automata have strictly less accepting power (even the nondeterministic variant). Nondeterministic Büchi automata can be used only for S1S.

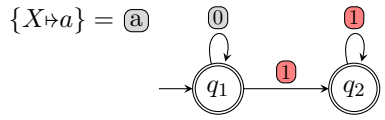


Figure 6.5: Muller automaton with the accepting condition $\{\{q_1\},\{q_2\}\}$ corresponding to an S1S formula $\varphi(X) \triangleq \exists Y.\ Y = S(X) \wedge Y \subseteq X$ expressing the fact that X contains all consecutive numbers from some $n \in \omega$ (or X is the empty set).

Table 6.1: (W)SkS expressiveness results.

Language L	**Equivalence**
Infinite words	L is WS1S-definable $\iff$ L is S1S-definable
	$\iff$ L is ω-regular
Finite words	L is WS1S-definable $\iff$ L is regular
Infinite binary trees	L is SkS-definable $\iff$ L is accepted by a Muller tree automaton
Finite binary trees	L is WSkS-definable $\iff$ L is a regular tree language

for the definition of Muller accepting condition). An example of an S1S formula with a corresponding automaton is shown in Figure 6.5.

Expressivity power. Although (W)SkS may seem from the definition of semantics as a weak logic (regarding its descriptional power), the opposite is true. In fact, it can be shown that a language of infinite words is ω-regular if and only if is (W)S1S-definable [65][2]. Moreover, for finite words, WS1S defines the class of regular languages and, for finite trees, WSkS defines the class of regular tree languages [66, 98, 268]. On top of that, every formula of Presburger arithmetic is equivalently expressible in WS1S. We mentioned that each ω-regular language can be defined using WS1S. This follows from the effective equivalence of nondeterministic and deterministic Muller automata and from the fact that an accepting run of a deterministic Muller automaton can be described using WS1S [242]. However, this is not possible for WS2S, because deterministic Muller tree automata are strictly weaker than their nondeterministic variants. An overview of the expressiveness results is shown in Table 6.1 (see [242] for more details).

[2]If we want to speak about equivalence of automata and logics, we need to extend (W)SkS by predicates representing symbols.

Chapter 7

Automata Terms in a Lazy WSkS Decision Procedure

In Chapter 6, we introduced WSkS and its decision procedures. WSkS offers extreme succinctness for the price of **NONELEMENTARY** worst-case complexity. However, as already described in Section 6.2.4, the trade-off between complexity and succinctness may be turned significantly favourable in many practical cases through a use of clever implementation techniques and heuristics improving the basic automata-based decision procedure. Such techniques, as already mentioned in the previous chapter, were then elaborated in the tool MONA, the best-known implementation of decision procedures for WS1S and WS2S. Despite the extensive research and engineering effort invested into MONA, it is, however, easy to reach its scalability limits. Particularly, MONA implements the classical automata-based decision procedure that builds a tree automaton representing models of the given formula and then checks emptiness of the automaton's language (see Section 6.2.3 for more details). The **NONELEMENTARY** complexity manifests in that the size of the automaton is prone to explode, which is caused mainly by repeated determinization (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 [196], or, often restricting the input of their approach, give up using WSkS altogether [289].

In this chapter, we propose a decision procedure for WS2S. Note that, the restrictions of two successors does not change the expressive power of the logic since k-ary trees, for $k > 2$, can be easily encoded into binary ones. We revisit the use of tree automata 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 [99, 292, 53, 14], 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* determinization 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). Our decision procedure described in this chapter extends and generalizes the approaches of on-the-fly automata construction, subsumption, and lazy evaluation for the needs of deciding WS2S.

Overview of the proposed approach. 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.

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 optimizations. 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 optimizations 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.

Related work. The related decision procedures aiming at WSkS (and not only) are mentioned in Section 6.2.3. Applications of WSkS are discussed in Section 6.2.4. We recall here the most relevant works for the contents of this chapter. The tool MONA [143, 105] implements the classical decision procedures for both WS1S and WS2S. It is still the standard tool of choice for deciding WS1S/WS2S 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.) [173, 171]. The decision procedure for the MSO(STR) logic was implemented within, e.g., JMOSEL [272] or within the symbolic finite automata framework of [90]. In particular, JMOSEL implements several optimizations allow it to outperform MONA on some benchmarks. A new decision procedure for the weak monadic second-order logic on inductive structures was developed in [121] within the tool TOSS. The presented 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 arithmetic and one formula of

the form that we mention in our experiments as problematic for MONA but solvable easily by MONA with antiprenexing (an optimization properly discussed in Chapter 8).

The original inspiration for this work are the antichain techniques for checking universality and inclusion of finite automata [99, 292, 53, 14] and language emptiness of alternating automata [99], which use symbolic computation together with subsumption to prune large state spaces arising from subset construction. Antichain algorithms and their generalizations have shown great efficiency improvements in applications such as abstract regular model checking [53], shape analysis [134], LTL model checking [293], or game solving [291].

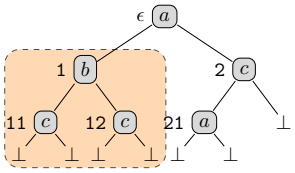
Our approach is a generalization of the works [112] and especially [111] on WS1S. Although the term structure and the generalized algorithm may seem close to [111], the reasoning behind it is significantly more involved. Particularly, [111] is based on defining the semantics (language) of terms as a 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 [111] 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.

Chapter outline. This chapter is organized as follows. Section 7.1 describes an automata-based decision procedure. Section 7.2 introduces the notion of automata terms and a decision procedure of WS2S based on them. Section 7.3 then presents various optimizations in the decision procedure. Section 7.4 deals with an experimental evaluation and finally Section 7.5 concludes the chapter.

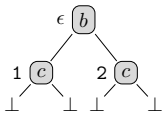
7.1 The Explicit Decision Procedure

In this section, we extend the definitions related to trees from Chapters 2 and 6. We also present a variant of the WS2S automata-based decision procedure considered in this chapter.

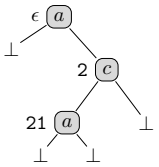
Basics and trees. In this chapter, for a binary operator $\bullet$, we write $A\,[\bullet]\,B$ to denote the augmented product $\{a \bullet b \mid (a, b) \in A \times B\}$ of A and B. Since we deal with WS2S in this chapter, we will consider ordered binary trees and binary tree automata. We let $\mathit{leaf}(\tau)$ be the set of all leaves of τ. 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 $S \subseteq \Sigma^*$ 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



(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 $\{\mu\}$

Figure 7.1: 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).

of τ obtained from τ by removing some of the sub-trees in S. The derivative of a set of trees $T \subseteq \Sigma^\star$ with respect to S is the set $\bigcup_{\tau \in T}(\tau - S)$. See Figure 7.1 for an example of the derivative.

The explicit decision procedure for WS2S. In this chapter, we use adjusted decision procedure from Section 6.2.3. In particular, we fix a formula φ over variables $\mathbb{X}$. The decision procedure then again constructs the automaton $\mathcal{A}_\varphi$ inductively to the structure of φ, as follows (note that we abuse the notation of $\mathcal{A}_\varphi$ from Section 6.2.3): (i) If φ is an atomic formula, then $\mathcal{A}_\varphi$ is a pre-defined *base* TA over whole $\Sigma_\mathbb{X}$. (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 $\mathcal{A}_\varphi = det(\pi_X(\mathcal{A}_\psi)) - \vec{0}^\star$. Since the base TAs are extended to symbols over $\mathbb{X}$, it is not necessary to perform cylindrification in the case of $\wedge$ and $\vee$. We further implicitly assume that the top-level automaton $\mathcal{A}_\varphi$ is $\pi^\flat$-projected to obtain alphabet containing only symbols over free variables of φ.

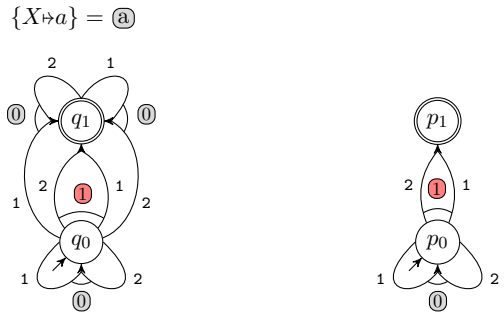
7.2 Automata Terms

Our algorithm for deciding WS2S may be seen as an alternative implementation of the classical procedure from Section 7.1. 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 Section 7.3.

7.2.1 Syntax of Automata Terms.

Terms are created according to the grammar

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



(a) $\mathcal{A}_{\mathrm{Sing}(X)}$ **(b)** $\mathcal{A}_{X=\{\epsilon\}}$

Figure 7.2: Tree automata for the predicates used in Example 7.2.1

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 $\vec{0}$'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 7.2.1. *Consider a formula* $\varphi \triangleq \neg\exists X.\ \mathrm{Sing}(X) \wedge X = \{\epsilon\}$ *and its corresponding automata term* $t_\varphi = \left\{ \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^*} \right\}$ *(we will show how* t_φ *was obtained from* φ *later). For the sake of presentation, we will consider the base automata given in Figure 7.2 for the predicates* $\mathrm{Sing}(X)$ *and* $X = \{\epsilon\}$*. The term* t_φ *above denotes the TA* $\left(det(\pi_X(\mathcal{A}_{\mathrm{Sing}(X)} \cap \mathcal{A}_{X=\{\epsilon\}})) - \vec{0}^*\right)^{\complement}$ *constructed using the automata operations of intersection, projection, subset construction, derivative, and complement.*

7.2.2 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 Figure 7.3a, and Δ is defined in Figure 7.3b, with the addition that whenever the rules in Figure 7.3b 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 Figures 7.3a and 7.3b, 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

$$\mathcal{R}(t+u) \Leftrightarrow \mathcal{R}(t) \vee \mathcal{R}(u) \quad (7.1)$$
$$\mathcal{R}(t \,\&\, u) \Leftrightarrow \mathcal{R}(t) \wedge \mathcal{R}(u) \quad (7.2)$$
$$\mathcal{R}(\pi_X(t)) \Leftrightarrow \mathcal{R}(t) \quad (7.3)$$
$$\mathcal{R}(\overline{t}) \Leftrightarrow \neg\mathcal{R}(t) \quad (7.4)$$
$$\mathcal{R}(S) \Leftrightarrow \exists t \in S.\, \mathcal{R}(t) \quad (7.5)$$
$$\mathcal{R}(q) \Leftrightarrow q \in R^{\mathcal{B}} \quad (7.6)$$

(a) Root term states

$$\Delta_a(t+u, t'+u') = \Delta_a(t,t')\,[+]\,\Delta_a(u,u') \quad (7.7)$$
$$\Delta_a(t \,\&\, u, t' \,\&\, u') = \Delta_a(t,t')\,[\&]\,\Delta_a(u,u') \quad (7.8)$$
$$\Delta_a(\pi_X(t), \pi_X(t')) = \{\pi_X(u) \mid u \in \Delta_{\pi_X(a)}(t,t')\} \quad (7.9)$$
$$\Delta_a(\overline{t}, \overline{t'}) = \{\overline{u} \mid u \in \Delta_a(t,t')\} \quad (7.10)$$
$$\Delta_a(S,S') = \left\{ \bigcup_{t\in S, t'\in S'} \Delta_a(t,t') \right\} \quad (7.11)$$
$$\Delta_a(q,r) = \delta_a^{\mathcal{B}}(q,r) \quad (7.12)$$

(b) Transitions among compatible state terms

Figure 7.3: Semantics of terms

product union, product intersection, projection, complement, and subset construction, respectively, build the transition function from the transition functions of their arguments (cf. Section 12.1). 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 $S - \vec{0}^{\star}$. This term is a symbolic representation of the set term that contains all state terms upward-reachable from S in $\mathcal{A}_S$ over $\vec{0}$. Formally, we first define the so-called *saturation* of $\mathcal{A}_S$ as

$$(S - \vec{0}^{\star})^{\mathsf{s}} = \mathit{reach}_{\Delta_{\vec{0}}}(S) \quad (7.13)$$

(with $\mathit{reach}_{\Delta_{\vec{0}}}(S)$ defined as the fixpoint (6.1)), and we complete the definition of Δ and $\mathcal{R}$ in Figures 7.3a and 7.3b with three new rules to be used with a derivative term D:

$$\Delta_a(D,u) = \Delta_a(D^{\mathsf{s}},u) \quad (7.14) \qquad \mathcal{R}(D) \Leftrightarrow \mathcal{R}(D^{\mathsf{s}}) \quad (7.16)$$
$$\Delta_a(u,D) = \Delta_a(u,D^{\mathsf{s}}) \quad (7.15)$$

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 7.2.2. *Let us consider a derivative term* $t = \{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^{\star}$*, which occurs within the nested automata term* t_φ *of Example 7.2.1. The set term representing all terms reachable upward from* t *is then the term*

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

The semantics of t *is then the automaton* $\mathcal{A}_t$ *with the set of states given by* t^{s}*.*

7.2.3 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 Section 7.3, is that the automata represented by terms indeed have finitely many states. This is a direct consequence of the following lemma.

Lemma 7.2.1. *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) $d(S - \Gamma^{\vec{\star}}) = 1 + d(S)$. 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) $t^{\mathsf{e}} = t$ if t is expanded and (ii) $t^{\mathsf{e}} = (t[u/u^{\mathsf{s}}])^{\mathsf{e}}$ where u is a derivative term of the form $S - \Gamma^{\vec{\star}}$ 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., $A^{\mathsf{e}} = \{t_1, \ldots, t_n\}$.

Lemma 7.2.2. *Given an automata term t and its expanded term t^{e}, it holds that*

(i) t^{e} is of a finite size and

(ii) $\mathcal{L}(t^{\mathsf{e}}) = \mathcal{L}(t)$.

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 (7.13) and Lemma 7.2.1, it follows that each such saturation is finite.

(ii): First, note that saturation preserves language, i.e., it holds that

$$\mathcal{L}\left((S - \vec{0}^{\vec{\star}})\right) = \mathcal{L}\left((S - \vec{0}^{\vec{\star}})^{\mathsf{s}}\right). \tag{7.17}$$

The previous fact follows from the definition of derivative automaton in Section 7.1. In particular, given $\mathcal{A}_S = (Q, \Delta, S, R)$, we have that

$$\mathcal{A}_S - \vec{0}^{\vec{\star}} = (Q, \Delta, reach_{\Delta_{\vec{0}}}(S), R), \tag{7.18}$$

which matches the definition of saturation in (7.13). The lemma follows from the fact that the expansion substitutes terms for saturated terms with equal languages. □

Lemma 7.2.3 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 [111], where every term has its own language, which makes the reasoning and the correctness proofs in the current work significantly more involved.

Lemma 7.2.3. *For automata terms A_1, A_2 and a set term S, the following equalities hold:*

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

Proof. (a): We prove the following more general form of (a):

$$\mathcal{L}\left(\{A_1, \ldots, A_n\}\right) = \mathcal{L}\left(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}\right). \tag{7.19}$$

(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 determinization 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 $\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}$.

$(\subseteq)$ Let τ be a tree. It holds that $\tau \in \mathcal{L}\left(\{A_1, \ldots, A_n\}\right)$ 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 determinization of TAs). Moreover, for all non-leaf positions $w \in \mathrm{dom}(\tau) \setminus \mathit{leaf}(\tau)$, let $\rho(w) = U, \rho(w.1) = U_1$, and $\rho(w.2) = U_2$. Then, from (7.11), we have that if $u \in U$, then there exist $u_1 \in U_1$ and $u_2 \in U_2$ such that $u \in \Delta_{\tau(w)}(u_1, u_2)$. Let us define an auxiliary function $\mu(w, u) = (u_1, u_2)$ 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 $\rho'(w.1) = u_1$ and $\rho'(w.2) = u_2$ where $(u_1, u_2) = \mu(w, \rho'(w))$. 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 $\rho'(w) \in \mathit{reach}_{\Delta}(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}})$ 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 $\tau \in \mathcal{L}\left(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}\right)$.

$(\supseteq)$ Consider a tree $\tau \in \mathcal{L}\left(\bigcup_{1 \leq i \leq n} A_i^{\mathsf{e}}\right)$. 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 $\{r\} = \Delta_{\tau(w)}(\rho'(w.1), \rho'(w.2))$ (we know that $\Delta_{\tau(w)}(\rho'(w.1), \rho'(w.2))$ is a singleton set due to (7.11)). 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., $\tau \in \mathcal{L}\left(\{A_1, \ldots, A_n\}\right)$.

(b): $(\subseteq)$ Let $\tau \in \mathcal{L}\left(\{A_1 + A_2\}\right)$. Then there is an accepting run ρ on τ in $\mathcal{A}_{\{A_1^{\mathsf{e}} + A_2^{\mathsf{e}}\}}$. 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 $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$ and $\mathcal{A}_{\{A_2^{\mathsf{e}}\}}$ 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, $\tau \in \mathcal{L}\left(\mathcal{A}_{\{A_1^{\mathsf{e}}\}}\right)$ or $\tau \in \mathcal{L}\left(\mathcal{A}_{\{A_2^{\mathsf{e}}\}}\right)$, so $\tau \in \mathcal{L}\left(\{A_1\}\right) \cup \mathcal{L}\left(\{A_2\}\right)$ and from (a) we get $\tau \in \mathcal{L}\left(A_1\right) \cup \mathcal{L}\left(A_2\right)$.

$(\supseteq)$ Consider $\tau \in \mathcal{L}\left(A_1\right) \cup \mathcal{L}\left(A_2\right)$. From (a) we get $\tau \in \mathcal{L}\left(\{A_1\}\right) \cup \mathcal{L}\left(\{A_2\}\right)$. Then there are runs ρ_1 in $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$ and ρ_2 in $\mathcal{A}_{\{A_2^{\mathsf{e}}\}}$ 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 $\mathcal{A}_{\{A_1^{\mathsf{e}} + A_2^{\mathsf{e}}\}}$. Therefore $\tau \in \mathcal{L}\left(\{A_1 + A_2\}\right)$.

(c): Dual to (b).

(d): Let τ be a tree. We will consider runs ρ and $\overline{\rho}$ of $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$ and $\mathcal{A}_{\{\overline{A_1^{\mathsf{e}}}\}}$ on τ, respectively. First, note that both runs exist, which is guaranteed by the presence of the universal sink state $\emptyset$, cf. Section 7.2.2. Second, note that the two runs are unique, since there is a single leaf state and the transition function is deterministic by (7.11). Further, from (7.10), 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 $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$ if and only if $\overline{\rho}$ is accepting in $\mathcal{A}_{\{\overline{A_1^{\mathsf{e}}}\}}$. As a consequence, $\tau \in L(\{\overline{A_1^{\mathsf{e}}}\})$ if and only if $\tau \notin L(\{A_1^{\mathsf{e}}\})$. From ($a$), we know that $L(\{A_1^{\mathsf{e}}\}) = L(A_1^{\mathsf{e}})$.

(e): ($\subseteq$) Let $\tau \in \mathcal{L}(\{\pi_X(A_1)\})$ and ρ be an accepting run of $\mathcal{A}_{\{\pi_X(A_1^{\mathsf{e}})\}}$ on τ. From the definition of the transition function in (7.9) and (7.3), we get that there is an accepting run ρ' on some τ' in $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$ 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 $\mathcal{A}_{\{A_1^{\mathsf{e}}\}}$. Then there is also an accepting run ρ' on τ in $\mathcal{A}_{\{\pi_X(A_1^{\mathsf{e}})\}}$ where $\forall w \in \mathrm{dom}(\tau) : \ \rho'(w) = \pi_X(\rho(w))$.

(f): We prove the following more general equality: $\mathcal{L}(S) - \Gamma^{\circledast} = \mathcal{L}\left(S - \Gamma^{\circledast}\right)$, for a set of symbols Γ (note that S is a set term). In the following text, given a set term U, we define $U \ominus \Gamma = U^{\mathsf{e}} \cup \bigcup\{\Delta_\Gamma(t_1, t_2) \mid t_1, t_2 \in U^{\mathsf{e}}\}$. Note that $\mathit{reach}_\Delta(U^{\mathsf{e}}) = \mathit{reach}_\Delta(U \ominus \Gamma)$. Further, we use $\Gamma^{\leq n}$ to denote the set of trees over Γ of height at most n, i.e., $\Gamma^{\leq n} = \{t \in \Gamma^{\circledast} \mid \forall w \in \mathrm{dom}(t) : \ |w| \leq n\}$. 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 $\rho(w) \notin U^{\mathsf{e}}$ (i.e., ρ maps w to a newly added leaf state) there exist $t_1^w, t_2^w \in U^{\mathsf{e}}$ such that $\rho(w) \in \Delta_\Gamma(t_1^w, t_2^w)$. 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 $\rho(w) \notin U^{\mathsf{e}}$, we define $\rho'(w.1) = t_1^w$ and $\rho'(w.2) = t_2^w$. The run ρ' is accepting in $\mathcal{A}_{U^{\mathsf{e}}}$ 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 $\mathcal{A}_{U^{\mathsf{e}}}$ on τ'. Consider the set $\Theta = \{w \in \mathit{leaf}(\tau) \mid \rho'(w) \notin U^{\mathsf{e}}\}$ of positions mapped by ρ' to newly added states. Since $\tau \in \tau' - \Gamma^{\leq 1}$, it holds that $\forall w \in \Theta : \ \rho'(w.1) \in U^{\mathsf{e}} \wedge \rho'(w.2) \in U^{\mathsf{e}} \wedge \tau'(w) \in \Gamma$. 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

$$\mathcal{L}(U_{m+1}) = \mathcal{L}(U_m \ominus \Gamma) = \mathcal{L}(U_m) - \Gamma^{\leq 1}. \tag{7.20}$$

  By the induction hypothesis we further have

$$\mathcal{L}(U_{m+1}) = (\mathcal{L}(U) - \Gamma^{\leq m}) - \Gamma^{\leq 1}. \tag{7.21}$$

Finally, from the definition of the derivative we obtain

$$(\mathcal{L}(U) - \Gamma^{\leq m}) - \Gamma^{\leq 1} = \mathcal{L}(U) - \Gamma^{\leq m+1}, \tag{7.22}$$

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 $S_0 = S^{\mathsf{e}}$ and $S_{i+1} = S_i \ominus \Gamma$. From the monotonicity of $\ominus$ and Lemma 7.2.1, 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 $\mathcal{L}(S_n) = \mathcal{L}(S) - \Gamma^{\star}$. Finally, we have $S_n = \mathit{reach}_{\Delta_\Gamma}(S^{\mathsf{e}}) = S - \Gamma^{\star}$ (by (7.13)), so we conclude that $\mathcal{L}(S) - \Gamma^{\star} = \mathcal{L}\left(S - \Gamma^{\star}\right)$. □

Lemma 7.2.3 shows fundamental properties of terms. Based on it we further focus on flattening of terms, whose properties are described by the following lemma.

Lemma 7.2.4. *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 $\tau \in \mathcal{L}(\{S + S'\})$. From Lemma 7.2.3$b$ we have $\mathcal{L}(\{S + S'\}) = \mathcal{L}(S) \cup \mathcal{L}(S')$. Hence there are runs ρ_1 in $\mathcal{A}_{S^{\mathsf{e}}}$ and ρ_2 in $\mathcal{A}_{S'^{\mathsf{e}}}$ 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 $\mathcal{A}_{\{t_1^{\mathsf{e}} + t_2^{\mathsf{e}} \mid t_1 \in S, t_2 \in S'\}}$ 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, $\tau \in \mathcal{L}(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\})$. From the definition of the augmented product, it follows that $\tau \in \mathcal{L}(S\,[+]\,S')$ and, finally, from Lemma 7.2.3a, we have $\tau \in \mathcal{L}(\{S\,[+]\,S'\})$.

$(\supseteq)$ Let $\tau \in \mathcal{L}(\{S\,[+]\,S'\})$. From Lemma 7.2.3$a$, we get $\tau \in \mathcal{L}(S\,[+]\,S')$, and from the definition of the augmented product, we obtain that $\tau \in \mathcal{L}(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\})$. 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 (7.11) 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 $\tau \in \mathcal{L}(\{S + S'\})$.

(b): Dual to (a).

(c): From Lemma 7.2.3e 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 $\mathcal{A}_{S^{\mathsf{e}}}$ on τ'. We will construct a run ρ' of $\mathcal{A}_{\{\pi_X(t) \mid t \in S^{\mathsf{e}}\}}$ 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 $\mathcal{A}_{\{\pi_X(t) \mid t \in S^{\mathsf{e}}\}}$ 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 $\tau \in \pi_X(\mathcal{L}(S))$. □

7.2.4 Terms of Formulae.

Our algorithm in Section 7.3 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:

$$\langle\varphi_0\rangle = I_{\varphi_0} \tag{7.23}$$
$$\langle\varphi \wedge \psi\rangle = \langle\varphi\rangle \,\&\, \langle\psi\rangle \tag{7.24}$$
$$\langle\varphi \vee \psi\rangle = \langle\varphi\rangle + \langle\psi\rangle \tag{7.25}$$
$$\langle\neg\varphi\rangle = \overline{\langle\varphi\rangle} \tag{7.26}$$
$$\langle\exists X.\ \varphi\rangle = \{\pi_X(\langle\varphi\rangle)\} - \vec{0}^* \tag{7.27}$$

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 determinization by subset construction, such double determinization terms can be always simplified to $\{t\}$ (cf. Lemma 7.2.3*a*). See Example 7.2.1 for a formula φ and its corresponding term t_φ. Theorem 7.2.1 establishes the correctness of the formula-to-term translation.

Theorem 7.2.1. *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:

$$U ::= \{u, \ldots, u\} \mid \{\pi_X(u), \ldots, \pi_X(u)\} \tag{7.28}$$
$$u ::= q \mid u + u \mid u \,\&\, u \mid \overline{u} \mid U \mid U - \Gamma^* \tag{7.29}$$

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 Section 7.2.1 (e.g., the term $\pi_X(t_1) \,\&\, \pi_X(t_2)$ 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 Section 7.1.

– $\varphi = \varphi_0$ where φ_0 is an atomic formula: Let I_{φ_0} be the set of leaf states of $\mathcal{A}_{\varphi_0}$.

$$\begin{aligned} \mathcal{L}(\{\langle\varphi_0\rangle\}) &= \mathcal{L}(\{I_{\varphi_0}\}) && \text{⟅(7.23)⟆} \\ &= \mathcal{L}(I_{\varphi_0}) && \text{⟅Lemma 7.2.3}a\text{⟆} \\ &= \mathcal{L}(\mathcal{A}_{\varphi_0}) && \text{⟅term semantics⟆} \end{aligned}$$

– $\varphi = \psi_1 \wedge \psi_2$: We use the following equational reasoning:

$$\begin{aligned}
\mathcal{L}(\{\langle \psi_1 \wedge \psi_2 \rangle\}) &= \mathcal{L}(\{\langle \psi_1 \rangle \,\&\, \langle \psi_2 \rangle\}) && \text{⟅(7.24)⟆} \\
&= \mathcal{L}(\{\{\langle \psi_1 \rangle \,\&\, \langle \psi_2 \rangle\}\}) && \text{⟅Lemma 7.2.3}a\text{⟆} \\
&= \mathcal{L}(\{\{\langle \psi_1 \rangle\} \,\&\, \{\langle \psi_2 \rangle\}\}) && \text{⟅Lemma 7.2.4}b\text{⟆} \\
&= \mathcal{L}(\{\langle \psi_1 \rangle\}) \cap \mathcal{L}(\{\langle \psi_2 \rangle\}). && \text{⟅Lemma 7.2.3}c\text{⟆} \\
&= \mathcal{L}(\mathcal{A}_{\psi_1}) \cap \mathcal{L}(\mathcal{A}_{\psi_2}) && \text{⟅induction hypothesis⟆} \\
&= \mathcal{L}(\mathcal{A}_{\varphi}). && \text{⟅classical procedure⟆}
\end{aligned}$$

– $\varphi = \psi_1 \vee \psi_2$: We use the following equational reasoning:

$$\begin{aligned}
\mathcal{L}(\{\langle \psi_1 \vee \psi_2 \rangle\}) &= \mathcal{L}(\{\langle \psi_1 \rangle + \langle \psi_2 \rangle\}) && \text{⟅(7.25)⟆} \\
&= \mathcal{L}(\{\{\langle \psi_1 \rangle + \langle \psi_2 \rangle\}\}) && \text{⟅Lemma 7.2.3}a\text{⟆} \\
&= \mathcal{L}(\{\{\langle \psi_1 \rangle\} + \{\langle \psi_2 \rangle\}\}) && \text{⟅Lemma 7.2.4}a\text{⟆} \\
&= \mathcal{L}(\{\langle \psi_1 \rangle\}) \cup \mathcal{L}(\{\langle \psi_2 \rangle\}). && \text{⟅Lemma 7.2.3}b\text{⟆} \\
&= \mathcal{L}(\mathcal{A}_{\psi_1}) \cup \mathcal{L}(\mathcal{A}_{\psi_2}) && \text{⟅induction hypothesis⟆} \\
&= \mathcal{L}(\mathcal{A}_{\varphi}). && \text{⟅classical procedure⟆}
\end{aligned}$$

– $\varphi = \neg\psi$: First, we prove the following claim:

Claim 3: *Let t be a deterministic term, then* $\mathcal{L}\left(\left\{\overline{\{t\}}\right\}\right) = \mathcal{L}(\{\bar{t}\})$.

Proof First, consider two expanded deterministic terms t_1 and t_2. Since t_1 and t_2 are deterministic, from (7.11) we have $\Delta_a(t_1, t_2) = \{t'\}$ for some deterministic term t' and any symbol a. Therefore (from (7.10)), $\Delta_a(\overline{t_1}, \overline{t_2}) = \{\overline{t'}\}$ 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 $\rho(w) = \overline{s} \Leftrightarrow \rho'(w) = \overline{\{s\}}$. ■

We proceed to the main part of the proof.

$$\begin{aligned}
\mathcal{L}(\{\langle \neg\psi \rangle\}) &= \mathcal{L}\left(\left\{\overline{\langle \psi \rangle}\right\}\right) && \text{⟅(7.26)⟆} \\
&= \mathcal{L}\left(\left\{\overline{\{\langle \psi \rangle\}}\right\}\right) && \text{⟅Claim 3⟆} \\
&= \overline{\mathcal{L}(\{\langle \psi \rangle\})} && \text{⟅Lemma 7.2.3}d\text{⟆} \\
&= \overline{\mathcal{L}(\mathcal{A}_{\psi})} && \text{⟅induction hypothesis⟆} \\
&= \mathcal{L}(\mathcal{A}_{\varphi}). && \text{⟅classical procedure⟆}
\end{aligned}$$

– $\varphi = \exists X.\ \psi$: 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 (7.9), $\Delta_a(\pi_X(t_1), \pi_X(t_2)) = \{\pi_X(t_b) \mid b \in \pi_X(a)\}$

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.

$$\begin{aligned}
\mathcal{L}(\{\langle \exists X.\ \psi \rangle\}) &= \mathcal{L}\left(\{\pi_X(\langle \psi \rangle)\} - \vec{0}^{\circledast}\right) && \text{⟅(7.27)⟆} \\
&= \mathcal{L}(\{\pi_X(\langle \psi \rangle)\}) - \vec{0}^{\circledast} && \text{⟅Lemma 7.2.3}f\text{⟆} \\
&= \mathcal{L}(\{\pi_X(\{\langle \psi \rangle\})\}) - \vec{0}^{\circledast} && \text{⟅Claim 4⟆} \\
&= \pi_X(\mathcal{L}(\{\langle \psi \rangle\})) - \vec{0}^{\circledast} && \text{⟅Lemma 7.2.3}e\text{⟆} \\
&= \pi_X(\mathcal{L}(\mathcal{A}_\psi)) - \vec{0}^{\circledast} && \text{⟅induction hypothesis⟆} \\
&= \mathcal{L}(\mathcal{A}_\varphi). && \text{⟅classical procedure⟆}
\end{aligned}$$

□

7.3 An Efficient Decision Procedure

The development in Section 7.2 already implies a naive automata term-based satisfiability check. Namely, by Theorem 7.2.1, 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 (7.23)–(7.27), we may use the definitions of the transition function and root states of $\mathcal{A}_{t_\varphi} = (Q, \Delta, I, F)$ in Section 7.2 to decide the language emptiness through evaluating the root state test $\mathcal{R}(\mathit{reach}_\Delta(I))$. The equalities and equivalences (7.7)–(7.16) can be implemented as recursive functions. We will further refer to this algorithm as the *simple recursion*. The evaluation of $\mathit{reach}_\Delta(I)$ induces nested evaluations of the fixpoint (7.13): 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 7.2.1.

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 Section 7.2 through *product flattening* and *nondeterministic union*, which are analogous to standard implementations of automata intersection and union.

7.3.1 Memoization

The simple recursion repeats the fixpoint computations that saturate derivative terms from scratch at every call of the transition function or root test. This is easily countered through *memoization*, known, e.g., from compilers of functional languages, which caches results of function calls in order to avoid their re-evaluation. Namely, after saturating a derivative sub-term $t = S - \vec{0}^{\circledast}$ of t_φ for the first time, we simply *replace* t in t_φ by the saturation $t^{\mathsf{s}} = \mathit{reach}_{\Delta_{\vec{0}}}(S)$. Since a derivative is a symbolic representation of its saturated version

(cf. (7.13)), the replacement does not change the language of t_φ. Using memoization, every fixpoint computation is then carried out only once.

7.3.2 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 $\mathcal{R}(S - \vec{0}^*)$ (obtained from sub-formulae such as $\exists X.\ \psi$). Namely, instead of first unfolding the whole fixpoint into a set $\{t_1, \ldots, t_n\}$ and only then testing whether $\mathcal{R}(t_i)$ is true for some t_i, the terms t_i can be tested as soon as they are computed, and the fixpoint computation can be stopped early, immediately when the test succeeds on one of them. Then, instead of replacing the derivative sub-term by its full saturation, we replace it by the partial result $\{t_1, \ldots, t_i\} - \vec{0}^*$ for $i \leq n$. Finishing the evaluation of the fixpoint computation might later be required in order to compute a transition from the derivative. We note that this corresponds to the concept of *continuations* from functional programming, used to represent a paused computation that may be required to continue later.

Example 7.3.1. *Let us now illustrate the lazy decision procedure on our running example formula* $\varphi \triangleq \neg\exists X.\ \mathrm{Sing}(X) \wedge X = \{\epsilon\}$ *and the corresponding automata term* $t_\varphi = \left\{ \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^*} \right\}$ *from Example 7.2.1. The task of the procedure is to compute the value of* $\mathcal{R}(\mathit{reach}_\Delta(t_\varphi))$*, i.e., whether there is a root state reachable from the leaf state* $\langle\varphi\rangle$ *of* $\mathcal{A}_{t_\varphi}$*. The fact that* φ *is ground allows us to slightly simplify the problem because any ground formula* ψ *is satisfiable 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* $\langle\varphi\rangle = \overline{\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^*}$.

The computation proceeds as follows. First, we use (7.4) *from Figure 7.3a to propagate the root test towards the derivative, i.e., to obtain that* $\mathcal{R}(\langle\varphi\rangle)$ *holds if and only if* $\neg\mathcal{R}(\{\pi_X(\{q_0\} \,\&\, \{p_0\})\} - \vec{0}^*)$*. 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* (7.3) *and evaluates as false on the left conjunct (through, in order,* (7.2)*,* (7.5)*, and* (7.6)*) since the state* q_0 *is not a root state. As a trivial example of short circuiting, we can skip evaluating* $\mathcal{R}$ *on the right conjunct* $\{p_0\}$ *and conclude that* $\mathcal{R}(t_0)$ *is false.*

The fixpoint computation then continues with the first iteration, computing the $\vec{0}$*-successors of the set* $\{t_0\}$*. We will obtain the set* $\Delta_{\vec{0}}(t_0, t_0) = \{t_0, t_1\}$ *with* $t_1 = \pi_X(\{q_1\} \,\&\, \{p_1\})$*. The test* $\mathcal{R}(t_1)$ *now returns true because both* q_1 *and* p_1 *are root states. With that, the fixpoint computation may terminate early, with the* $\mathcal{R}$*-test on the derivative sub-term returning true. Memoization then replaces the derivative sub-term in* $\langle\varphi\rangle$ *by the partially evaluated version* $\{t_0, t_1\} - \vec{0}^*$*, and* $\mathcal{R}(\langle\varphi\rangle)$ *is evaluated as false due to* (7.4)*. We therefore conclude that* φ *is unsatisfiable (and invalid since it is ground).*

7.3.3 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 it is done for WS1S in the work [111]) the concept used in *antichain* algorithms for efficiently deciding language inclusion and universality of finite word and tree automata [99, 292, 53, 14]. 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:

$$\{t_1, t_2, \ldots, t_n\} \rightsquigarrow \{t_2, \ldots, t_n\} \qquad \text{for } t_1 \sqsubseteq t_2, \tag{7.30}$$

which may be used to simplify set sub-terms of automata terms. The rule (7.30) 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

$$S \sqsubseteq S' \quad \Leftrightarrow S \subseteq S' \vee S \sqsubseteq^{\forall\exists} S' \tag{7.31}$$
$$t \,\&\, u \sqsubseteq t' \,\&\, u' \quad \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{7.32}$$
$$t + u \sqsubseteq t' + u' \quad \Leftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u' \tag{7.33}$$
$$\overline{t} \sqsubseteq \overline{t'} \quad \Leftrightarrow t' \sqsubseteq t \tag{7.34}$$
$$\pi_X(t) \sqsubseteq \pi_X(t') \quad \Leftrightarrow t \sqsubseteq t' \tag{7.35}$$

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 (7.31)). 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 [14]. Consequently, rewriting sub-terms in an automata term according to the new rule (7.30) does not change its language.

7.3.4 Product Flattening

Product flattening is a technique that we use to reduce the size of fixpoint saturations that generate conjunctions and disjunctions of sets as their elements. Consider a term of the form $D = \{\pi_X(S_0 \,\&\, S_0')\} - \vec{0}^*$ 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. Section 7.1). 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*: $D = \{\pi_X(S_0 \,[\&]\, S_0')\} - \vec{0}^*$ where

[&] is the augmented product for conjunction. This way, we can decrease the worst-case size of the fixpoint to $|Q| \cdot |Q'|$. A similar reasoning holds for terms of the form $\{\pi_X(S_0 + S'_0)\} - \vec{0}^{\circledast}$. Formally, the technique can be implemented by the following pair of sub-term rewriting rules where S and S' are non-empty sets of terms:

$$S + S' \rightsquigarrow S \,[+]\, S', \qquad (7.36) \qquad\qquad S \,\&\, S' \rightsquigarrow S \,[\&]\, S'. \qquad (7.37)$$

Observe that for terms obtained from WS2S formulae using the translation from Section 7.2, the rules are not helpful in their given form. Consider, for instance, the term $\{\pi_X(\{r\} \,\&\{q\})\} - \vec{0}^{\circledast}$ obtained from a formula $\exists X. \varphi \wedge \psi$ with φ and ψ being atoms. The term would be, using rule (7.37), rewritten into the term $\{\pi_X(\{r \,\&\, q\})\} - \vec{0}^{\circledast}$. Then, during a subsequent fixpoint computation, we might obtain a fixpoint of the following form: $\{\pi_X(\{r \,\&\, q\}), \pi_X(\{r \,\&\, q, r_1 \,\&\, q_1\}),\ \pi_X(\{r_1 \,\&\, q_1, r_2 \,\&\, q_2\})\}$, where the occurrences of the projection π_X disallow one to perform the desired union of the inner sets, and so the application of rule (7.37) 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:

$$\pi_X(S) \rightsquigarrow \{\pi_X(t) \mid t \in S\}. \qquad (7.38)$$

In the example above, using rule (7.38) we would now obtain the term $\{\pi_X(r \,\&\, q)\} - \vec{0}^{\circledast}$ (note that we rewrote $\{\{\cdot\}\}$ to $\{\cdot\}$ as mentioned in Section 7.2) and the fixpoint $\{\pi_X(r \,\&\, q), \pi_X(r_1 \,\&\, q_1), \pi_X(r_2 \,\&\, q_2)\}$. The correctness of the rules is given by Lemma 7.2.4.

We, however, still have to note that there is a danger related with the rules (7.36)–(7.38). Namely, if they are applied to some terms in a partially evaluated fixpoint but not to all, the form of these terms might get different (cf. $\pi_X(\{r \,\&\, q\})$ and $\pi_X(r \,\&\, q)$), and it will not be possible to combine them as source states of TA transitions when computing Δ_a, leading thus to an incorrect result. We resolve the situation 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.

7.3.5 Nondeterministic Union

Optimization of the product term saturations from the previous section can be pushed one step further for terms of the form $\{\pi_X(S + S')\} - \vec{0}^{\circledast}$. The idea is to use the *nondeterministic (disjoint) 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:

$$S + S' \rightsquigarrow S \cup S' \qquad \text{for } S \not\bowtie S' \qquad (7.39)$$

where S and S' are sets of terms (similarly to Section 7.3.4, in order to successfully reduce the fixpoint state space on terms obtained from WS2S formulae, we also need to apply rule (7.38) 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:

$$S \bowtie S' \quad \Leftrightarrow S = S' \vee \exists t \in S, t' \in S' : t \bowtie t' \tag{7.40}$$

$$t \,\&\, u \bowtie t' \,\&\, u' \quad \Leftrightarrow t \bowtie t' \vee u \bowtie u' \tag{7.41}$$

$$t + u \bowtie t' + u' \quad \Leftrightarrow t \bowtie t' \vee u \bowtie u' \tag{7.42}$$

$$\bar{t} \bowtie \overline{t'} \quad \Leftrightarrow t \bowtie t' \tag{7.43}$$

$$\pi_X(t) \bowtie \pi_X(t') \quad \Leftrightarrow t \bowtie t' \tag{7.44}$$

$$D \bowtie t \quad \Leftrightarrow D^{\mathsf{s}} \bowtie t \tag{7.45}$$

$$t \bowtie D \quad \Leftrightarrow t \bowtie D^{\mathsf{s}} \tag{7.46}$$

$$q \bowtie r \quad \Leftrightarrow \exists 1 \leq k \leq n : q, r \in Q_k \tag{7.47}$$

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 (7.39), it is necessary to modify treatment of the sink state $\emptyset$ in the definition of Δ of Section 7.2. 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 $D = \{\pi_X(\{\bar{t}\} + \{\bar{r}\})\} - \vec{0}^{\star}$ such that $\Delta_{\vec{0}}(t, t) = \{t\}$, $\Delta_{\vec{0}}(r, r) = \{r\}$, and $\mathcal{R}(t)$ and $\mathcal{R}(r)$ are both *true*, i.e., both t and r can accept any $\vec{0}$-tree, which also means that the union of their complements should not accept any $\vec{0}$-tree. Indeed, the saturation of D is the set term $D^{\mathsf{s}} = \mathit{reach}_{\Delta_{\vec{0}}}(\{\pi_X(\{\bar{t}\} + \{\bar{r}\})\}) = \{\pi_X(\{\bar{t}\} + \{\bar{r}\})\}$ where it holds that $\neg\mathcal{R}(\pi_X(\{\bar{t}\} + \{\bar{r}\}))$, i.e., it does not accept any $\vec{0}$-tree. On the other hand, if we use the new rule (7.39) together with rule (7.38), we obtain the term $\{\pi_X(\bar{t}), \pi_X(\bar{r})\} - \vec{0}^{\star}$. When computing its saturation, we will obtain a new element $\Delta_{\vec{0}}(\pi_X(\bar{t}), \pi_X(\bar{r})) = \pi_X(\bar{\emptyset})$. The term $\pi_X(\bar{\emptyset})$ was constructed using the implicit rule of Section 7.2 that sends the otherwise undefined successors of a pair of terms to $\{\emptyset\}$. Note that $\mathcal{R}(\pi_X(\bar{\emptyset}))$ is *true*, yielding that the fixpoint approximation $\{\pi_X(\bar{t}), \pi_X(\bar{r}), \pi_X(\bar{\emptyset})\}$ is a root state, so a $\vec{0}$-tree is accepted. Therefore, the application of the new rule (7.39) changed the language.

Although the previous situation cannot happen with terms obtained from WS2S formulae using the translation rules from Section 7.2, 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 Section 7.2. We give the modified transition function $\Delta^\sharp$ in Figure 7.4.

Note that in the previous example, when using the modified transition function $\Delta^\sharp$ for computing the saturation of the term $\{\pi_X(\bar{t}), \pi_X(\bar{r})\} - \vec{0}^{\star}$, we would from $t \not\bowtie r$ deduce that $\pi_X(\bar{t}) \not\bowtie \pi_X(\bar{r})$. As a consequence, $\Delta^\sharp_{\vec{0}}(\pi_X(\bar{t}), \pi_X(\bar{r})) = \{\emptyset\}$, 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 Section 7.2 under the original semantics hold also for the modified semantics.

$$\Delta_a^\sharp(t,t') = \begin{cases} \Delta_a^\bullet(t,t') & \text{if } t \bowtie t' \\ \{\emptyset\} & \text{otherwise} \end{cases} \tag{7.48}$$

$$\Delta_a^\bullet(t+u,t'+u') = \Delta_a^\sharp(t,t')\,[+]\,\Delta_a^\sharp(u,u') \tag{7.49}$$

$$\Delta_a^\bullet(t \,\&\, u,t' \,\&\, u') = \Delta_a^\sharp(t,t')\,[\&]\,\Delta_a^\sharp(u,u') \tag{7.50}$$

$$\Delta_a^\bullet(\pi_X(t),\pi_X(t')) = \left\{ \pi_X(u) \;\middle|\; u \in \Delta^\sharp_{\pi_X(a)}(t,t') \right\} \tag{7.51}$$

$$\Delta_a^\bullet(\overline{t},\overline{t'}) = \left\{ \overline{u} \;\middle|\; u \in \Delta_a^\sharp(t,t') \right\} \tag{7.52}$$

$$\Delta_a^\bullet(S,S') = \left\{ \bigcup_{t\in S, t'\in S'} \Delta_a^\sharp(t,t') \right\} \tag{7.53}$$

$$\Delta_a^\bullet(q,r) = \delta_a^{\mathcal{B}}(q,r) \tag{7.54}$$

Figure 7.4: Modified transition function

Lemma 7.3.1. *For automata terms A_1, A_2 and a set term S, the following equalities hold:*

$$\mathcal{L}^\sharp(\{A_1\}) = \mathcal{L}^\sharp(A_1) \quad (a) \qquad \mathcal{L}^\sharp\left(\{\overline{A_1}\}\right) = \overline{\mathcal{L}^\sharp(A_1)} \quad (d)$$

$$\mathcal{L}^\sharp(\{A_1 + A_2\}) = \mathcal{L}^\sharp(A_1) \cup \mathcal{L}^\sharp(A_2) \quad (b) \qquad \mathcal{L}^\sharp(\{\pi_X(A_1)\}) = \pi_X(\mathcal{L}^\sharp(A_1)) \quad (e)$$

$$\mathcal{L}^\sharp(\{A_1 \,\&\, A_2\}) = \mathcal{L}^\sharp(A_1) \cap \mathcal{L}^\sharp(A_2) \quad (c) \qquad \mathcal{L}^\sharp\left(S - \vec{0}^*\right) = \mathcal{L}^\sharp(S) - \vec{0}^* \quad (f)$$

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):

$$\mathcal{L}^\sharp(\{A_1,\ldots,A_n\}) = \mathcal{L}^\sharp\left(\bigcup_{1\le i\le n} A_i^{\mathsf{e}}\right) \tag{7.55}$$

(Again, note that all expanded terms are set terms.) Intuitively, in this proof we show that determinization does not change the modified language of a term. Let us use $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ to denote the TA represented by the term $\bigcup_{1\le i\le n} A_i^{\mathsf{e}}$. 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 holds $\tau \in \mathcal{L}^\sharp(\{A_1^{\mathsf{e}},\ldots,A_n^{\mathsf{e}}\})$, 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 determinization 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 $\mathcal{A}_{\bigcup A_i^{\mathsf{e}}}$ 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 $\rho(w.1) = U_1$ and $\rho(w.2) = U_2$. Then, let $u_1 \in U_1$ and $u_2 \in U_2$ be terms such that $u \in \Delta^\sharp(u_1,u_2)$ (the presence of such terms is guaranteed by (7.53)). The following inductive invariant holds: If $u \neq \emptyset$, then $u_1 \neq \emptyset$

and $u_2 \neq \emptyset$ (the invariant follows from (7.53), the fact that $r \neq \emptyset$, and (7.40)). We set $\rho'(w.1) = u_1$ and $\rho'(w.2) = u_2$.

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 (7.53). 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 $\{r\} = \Delta^\sharp_{\tau(w)}(\rho'(w.1), \rho'(w.2))$ (we know that $\Delta^\sharp_{\tau(w)}(\rho'(w.1), \rho'(w.2))$ is a singleton set due to (7.53)). 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 7.2.3 (with altered term transition function).

(*f*): The proof is similar to the proof of Lemma 7.2.3*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 (7.47), 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 (7.41), it follows that either $u_1 \bowtie u_2$ or $v_1 \bowtie v_2$.

$$\begin{aligned}
\Delta^\sharp_a(t_1, t_2) &= \Delta^\sharp_a(u_1 \,\&\, v_1, u_2 \,\&\, v_2) \\
&= \Delta^\bullet_a(u_1 \,\&\, v_1, u_2 \,\&\, v_2) && \wr(7.48)\wr \\
&= \Delta^\sharp_a(u_1, u_2)\,[\&]\,\Delta^\sharp_a(v_1, v_2) && \wr(7.50)\wr \\
&= \{u \,\&\, v \mid u \in \Delta^\sharp_a(u_1, u_2) \wedge v \in \Delta^\sharp_a(v_1, v_2)\} && \wr\text{def. of } [\&]\wr
\end{aligned}$$

  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 (7.41), 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 (7.41), 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 7.3.2. *For sets of terms S and S' such that $S \neq \emptyset$ and $S' \neq \emptyset$, we have:*

$$\mathcal{L}^\sharp\left(\{S + S'\}\right) = \mathcal{L}^\sharp\left(\{S\,[+]\,S'\}\right), \tag{a}$$

$$\mathcal{L}^\sharp\left(\{S \,\&\, S'\}\right) = \mathcal{L}^\sharp\left(\{S\,[\&]\,S'\}\right), \tag{b}$$

$$\mathcal{L}^\sharp\left(\{\pi_X(S)\}\right) = \mathcal{L}^\sharp\left(\{\pi_X(t) \mid t \in S\}\right). \tag{c}$$

Proof. (a): $(\subseteq)$ Let $\tau \in \mathcal{L}^\sharp(\{S + S'\})$. From Lemma 7.3.1$b$ we have $\mathcal{L}^\sharp(\{S + S'\}) = \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp(S')$. 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, $\tau \in \mathcal{L}^\sharp(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\})$. From the definition of the augmented product, it follows that $\tau \in \mathcal{L}^\sharp(S\,[+]\,S')$ and, finally, from Lemma 7.3.1a, we have $\tau \in \mathcal{L}^\sharp(\{S\,[+]\,S'\})$.

$(\supseteq)$ Let $\tau \in \mathcal{L}^\sharp(\{S\,[+]\,S'\})$. From Lemma 7.3.1$a$, we get $\tau \in \mathcal{L}^\sharp(S\,[+]\,S')$, and from the definition of the augmented product, we obtain that $\tau \in \mathcal{L}^\sharp(\{t_1 + t_2 \mid t_1 \in S, t_2 \in S'\})$. 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 (7.11), 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 $\tau \in \mathcal{L}^\sharp(\{S + S'\})$.

(b): Dual to (a).

(c): Identical to the proof of Lemma 7.2.4c (with the altered transition function). □

The following theorem shows that formula-to-term translation is correct even for the modified semantics.

Theorem 7.3.1. *Let φ be a WS2S formula. Then, $\mathcal{L}^\sharp(t_\varphi) = \mathcal{L}(\varphi)$.*

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 Section 7.2.2. In the first step we prove $\mathcal{L}^\sharp(t_\psi) = \mathcal{L}(t_\psi)$ 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.

$$\begin{aligned}
\langle\varphi\rangle^{\mathsf{e},\Delta} = \langle\psi_1 \wedge \psi_2\rangle^{\mathsf{e},\Delta} &= (\langle\psi_1\rangle \,\&\, \langle\psi_2\rangle)^{\mathsf{e},\Delta} && \text{⟅(7.24)⟆} \\
&= \langle\psi_1\rangle^{\mathsf{e},\Delta} \,\&\, \langle\psi_2\rangle^{\mathsf{e},\Delta} && \text{⟅expansion propagation⟆} \\
&= \langle\psi_1\rangle^{\mathsf{e},\Delta^\sharp} \,\&\, \langle\psi_2\rangle^{\mathsf{e},\Delta^\sharp} && \text{⟅induction hypothesis⟆} \\
&= (\langle\psi_1\rangle \,\&\, \langle\psi_2\rangle)^{\mathsf{e},\Delta^\sharp} && \text{⟅expansion propagation⟆} \\
&= \langle\varphi\rangle^{\mathsf{e},\Delta^\sharp} && \text{⟅(7.24)⟆}
\end{aligned}$$

– $\varphi = \psi_1 \vee \psi_2$: Dual to $\psi_1 \wedge \psi_2$.

– $\varphi = \neg\psi$: We use the following equational reasoning.

$$\begin{aligned}
\langle\varphi\rangle^{\mathsf{e},\Delta} &= (\langle\overline{\psi}\rangle)^{\mathsf{e},\Delta} && \text{⟅(7.26)⟆} \\
&= \overline{\langle\psi\rangle}^{\mathsf{e},\Delta} && \text{⟅expansion propagation⟆} \\
&= \overline{\langle\psi\rangle}^{\mathsf{e},\Delta^\sharp} && \text{⟅induction hypothesis⟆} \\
&= (\langle\overline{\psi}\rangle)^{\mathsf{e},\Delta^\sharp} && \text{⟅expansion propagation⟆} \\
&= \langle\varphi\rangle^{\mathsf{e},\Delta^\sharp} && \text{⟅(7.26)⟆}
\end{aligned}$$

– $\varphi = \exists X.\ \psi$: We use the following equational reasoning.

$$\begin{aligned}
\langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta} &= (\{\pi_X(\langle\psi\rangle)\} - \vec{0}^*)^{\mathsf{e},\Delta} && \text{⟅(7.27)⟆} \\
&= \left(\mathit{reach}_{\Delta_{\vec{0}}}(\{\pi_X(\langle\psi\rangle)\})\right)^{\mathsf{e},\Delta} && \text{⟅(7.13)⟆} \\
&= \mathit{reach}_{\Delta_{\vec{0}}}\left(\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta})\}\right) && \text{⟅expansion propagation⟆} \\
&= \mathit{reach}_{\Delta_{\vec{0}}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) && \text{⟅induction hypothesis⟆}
\end{aligned}$$

From the inductive construction of $\langle\varphi\rangle$ let us now observe that for every $t_1, t_2 \in \mathit{reach}_{\Delta_{\vec{0}}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right)$ 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

$$\begin{aligned}
\langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta} &= \mathit{reach}_{\Delta_{\vec{0}}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) \\
&= \mathit{reach}_{\Delta^\sharp_{\vec{0}}}\left(\left\{\pi_X(\langle\psi\rangle^{\mathsf{e},\Delta^\sharp})\right\}\right) && \text{⟅previous reasoning⟆} \\
&= \langle\exists X.\ \psi\rangle^{\mathsf{e},\Delta^\sharp} && \text{⟅expansion prop. and (7.27)⟆}
\end{aligned}$$

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 $\mathcal{L}(t_\varphi) = \mathcal{L}^\sharp(t_\varphi)$. Finally, from Theorem 7.2.1 we have $\mathcal{L}(t_\varphi) = \mathcal{L}(\varphi)$, which concludes the proof. □

Based on Lemma 7.3.1, Lemma 7.3.2, and Theorem 7.3.1 we can show correctness of the nondeterministic union rule (7.39):

Lemma 7.3.3. *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 7.3.1*b*, we have $\mathcal{L}^\sharp(\{S + S'\}) = \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp(S')$. Let $\tau \in \mathcal{L}^\sharp(S) \cup \mathcal{L}^\sharp(S')$ and ρ be an accepting run on τ of either $\mathcal{A}_{S^\mathsf{e}}$ or $\mathcal{A}_{S'^\mathsf{e}}$. Therefore, ρ is an accepting run on τ also in $\mathcal{A}_{S^\mathsf{e} \cup S'^\mathsf{e}}$.

($\supseteq$) Let $\tau \in \mathcal{L}^\sharp(S \cup S')$. For each $t_1 \in S^\mathsf{e}$ and $t_2 \in S'^\mathsf{e}$ 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 $\mathcal{A}_{S^\mathsf{e} \cup S'^\mathsf{e}}$ on τ, then ρ is an accepting run on τ in either $\mathcal{A}_{S^\mathsf{e}}$ or $\mathcal{A}_{S'^\mathsf{e}}$. Without loss of generality, suppose that ρ is an accepting run on τ of $\mathcal{A}_{S^\mathsf{e}}$ 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 $\tau \in \mathcal{L}^\sharp(\{S + S'\})$. □

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 (7.39) 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.

7.4 Experimental Evaluation

We have implemented the above introduced techniques (with the exception of Section 7.3.5 for the reasons described therein) in a prototype Haskell tool.[1] The base automata, hard-coded into the tool, were the TAs for the basic predicates from Section 12.1, together with automata for predicates $\mathrm{Sing}(X)$ and $X = \{p\}$ for a variable X and a fixed tree position p. As an additional optimization, our tool uses the so-called *antiprenexing* (proposed already in [111]), 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 realized—can significantly help Mona too). Our preliminary implementation of product flattening (cf. Section 7.3.4) 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 optimizations built into Mona (for instance, for the benchmarks from [195], Mona on average took 0.1 s, while we timeouted). Next, we identified several parametric families of formulae (adapted from [111]), such as, e.g.,

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

Table 7.1: Experimental results over the following parametric families of formulae:

1. $\varphi_n^{pt} \triangleq \forall Z_1, Z_2.\ \exists X_1, \ldots, X_n.\ edge(Z_1, X_1) \wedge \bigwedge_{i=1}^{n} edge(X_i, X_{i+1}) \wedge edge(X_n, Z_2)$ where
   - $edge(X, Y) \triangleq edge_{\mathtt{1}}(X, Y) \vee edge_{\mathtt{2}}(X, Y)$ and
   - $edge_{\mathtt{1/2}}(X, Y) \triangleq \exists Z.\ Z = S_{\mathtt{1/2}}(X) \wedge Z \subseteq Y$
2. $\varphi_n^{cnst} \triangleq \exists X.\ X = \{(\mathtt{12})^4\} \wedge X = \{(\mathtt{12})^n\}$
3. $\varphi_n^{sub} \triangleq \forall X_1, \ldots, X_n\ \exists X.\ \bigwedge_{i=1}^{n-1} X_i \subseteq X \rightarrow (X_{i+1} = S_{\mathtt{1}}(X) \vee X_{i+1} = S_{\mathtt{2}}(X))$

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	–	–	2,487	–	–
	4	34.95	–	–	8,391	–	–
	5	60.94	–	–	23,827	–	–
φ_n^{cnst}	80	14.60	40.07	40.05	1,146	27,913	27,913
	90	21.03	64.26	64.20	1,286	32,308	32,308
	100	28.57	98.42	98.91	1,426	36,258	36,258
	110	38.10	–	–	1,566	–	–
	120	49.82	–	–	1,706	–	–
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	–	–	2,376	–	–

$\varphi_n^{horn} \triangleq \exists X.\ \forall X_1.\ \exists X_2, \ldots X_n.\ ((X_1 \subseteq X \wedge X_1 \neq X_2) \rightarrow X_2 \subseteq X) \wedge \ldots \wedge ((X_{n-1} \subseteq X \wedge X_{n-1} \neq X_n) \rightarrow X_n \subseteq X)$, 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 7.1, 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 7.1 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.

7.5 Conclusion

In this chapter, we introduced a novel decision procedure for WS2S based on on-demand evaluation of automata terms. We proposed several optimizations reducing the amount of generated state space. Our experimental evaluation shows a potential of our approach. Further improvements can include more performance-oriented implementation of the tool and more precise investigation of the combination of bottom-up (i.e., classical automata-based) and top-down (i.e., based on the automata terms) approach yielding a procedure that could push the results even further. This chapter was published in the proceedings of CADE-27 [138] and an extended version of this paper was accepted to appear in the Journal of Automated Reasoning [137].

Chapter 8

Antiprenexing for WSkS

As we already mentioned in Chapters 6 and 7, the WSkS logic provides a concise expression of regular tree properties. Despite the **NONELEMENTARY** complexity, WSkS found numerous applications across computer science also due to the tool MONA implementing the automata-based decision procedure. Although there have appeared several newer approaches and prototype tools that may beat MONA on restricted sets of formulae [272, 199, 273, 121, 111, 112], including the approach presented in Chapter 7 based on automata terms, MONA is still the most robust tool and handles by far the largest class of practical formulae. In this chapter, we focus on further improving the efficiency of MONA. Namely, we elaborate on the preprocessing technique known as *antiprenexing*, which pushes quantifiers deeper into a formula, narrowing their scope. We develop a formula preprocessing technique tuned specifically for MONA (although the approach is, in principle, applicable to all automata-based WSkS solvers).

Antiprenexing is advantageous for the satisfiability test of MONA for the following reason. Recall that MONA builds an automaton representing all models of the formula and then tests emptiness of its language. An automaton is built for every sub-formula, inductively to the structure of the formula, starting from predefined atomic automata for atomic formulae and using automata operations that model logical connectives to combine automata for sub-formulae to automata for larger formulae (see Section 6.2.3 for more details). The bottleneck is the size of the automata built during the process, which may grow with every automata operation, leading, in the worst case, to a tower of exponentials. For MONA, the logical connective with the most expensive automata counterpart is quantification, which involves determinization and is, therefore, exponential in the worst case. Antiprenexing pushes quantifiers deeper in the formula, and this causes that the costly quantification is applied on formulae with (hopefully) smaller automata that appear closer to atoms.

Overview of the proposed approach. Our formula preprocessing is implemented as a set of *syntactic rewriting rules*, most of which are well-known rules (or variants of rules) from transformations to the negation normal form, prenex normal form, or disjunctive normal form. The rules may, however, be applied in different ways and extent, and, if used in an unsuitable way, they may cause an explosion in the size of the formula and the automata built while deciding it. This can happen, e.g., due to unrestricted distribution of disjunctions over conjunctions, which may lead to an exponential growth of the formula, which would outweigh all potential benefits. To resolve the issue, we use *informed rules* that allow us to control the transformations based on how they change the *cost* of deciding the formula, which is given by the size of all automata to be constructed during the decision

procedure. Since we, of course, cannot construct the automata beforehand to get their precise size, we *estimate* their sizes using a *linear model* trained from runs of the decision procedure on various formulae using *linear regression*.

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

We demonstrate on a quite comprehensive benchmark that our formula preprocessing significantly improves the overall efficiency of MONA. Indeed, it allows MONA to solve several formulae of practical interest that, prior to our work, were beyond capabilities of any WSkS solver (including MONA).

Related work. The works related to various decision procedures of WSkS and its applications were properly discussed in Chapter 7 and Section 6.2. Although there are alternative methods outperform MONA on certain classes of formulae, we focus to improve MONA as it is still substantially most robust by a large margin (partially owing to the relative immaturity of the alternative tools).

Variants of our antiprenexing techniques presented in this chapter would certainly be relevant for the approaches [111, 112, 273] including the approach in Chapter 7. In [111, 112], a simple variant of antiprenexing was used (namely the quantifier distribution rule (QuantDistr)). Our more advanced techniques that use other rules according to a cost estimate cannot, however, be used directly. One would have to come up with different strategies and cost estimation techniques specific to these algorithms. For this reason, and also because most of the formulae from our benchmark are beyond the reach of other tools than MONA, we do not consider them in our experimental evaluation.

Basic principles of the transformation to antiprenex (or miniscope) form are a well-known folklore in theorem proving, QBF, and SMT solving. Its values were recognized, for instance, in [104, 44], and its origins reach at least to [285].

Chapter outline. This chapter is organized as follows. In Section 8.1, we describe the decision procedure implemented in MONA in more details. Then in Section 8.2, we present the proposed formula transformations in order to obtain smaller intermediate automata. Section 8.3 deals with an estimation of sizes of automata corresponding to WSkS formulae. In Section 8.4, we deal with the experimental evaluation and Section 8.5 concludes the chapter.

8.1 The Decision Procedure for WSkS in Mona

In this section, we briefly describe the WSkS decision procedure as implemented in the tool MONA. In the rest of this chapter, we further denote the set of free variables of a WSkS formula φ by $fv(\varphi)$ and the set of all sub-formulae of φ (including φ) by $sf(\varphi)$.

We now recall the variant of the classical decision procedure for WSkS implemented in MONA. Compared to the decision procedure described in Section 6.2.3, MONA is specific mainly in that it works with complete deterministic automata only and uses DTA minimization extensively. A DTA is *minimal* if it is complete and there is no complete DTA

with strictly less states that accepts the same language. To obtain the (canonical) minimal DTA it is possible to use straightforward generalizations of DFA minimization described in Section 3.3.1. We denote the minimal automaton equivalent to $\mathcal{A}$ by $min(\mathcal{A})$. We note that MONA uses a number of other crucial optimizations, such as a symbolic, BDD-based representation of the transition relation or the so-called three-valued semantics for automata with "don't care" states [173], but these are not directly relevant to the contribution of this chapter, and so we will not discuss them in this text.

When testing satisfiability of a formula φ, MONA constructs the minimal DTA $\mathcal{A}_\varphi$ over the alphabet $\Sigma_{fv(\varphi)}$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and then tests whether $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$. Recall that the emptiness test is implemented using the reachability analysis mentioned in Section 6.2.3.

The automaton $\mathcal{A}_\varphi$ is constructed by induction to the structure of φ. Note that we abuse the notation of $\mathcal{A}_\varphi$ from Section 6.2.3. Namely, if φ is an atomic formula with free variables $\mathbb{X}$, then $\mathcal{A}_\varphi$ is a pre-defined *base* minimal complete DTA over $\Sigma_\mathbb{X}$ (see Section 6.2.3 for more details). Otherwise, if φ is not atomic, then $\mathcal{A}_\varphi = min(\mathcal{A}^\partial_\varphi)$, i.e., it is obtained by minimizing the DTA $\mathcal{A}^\partial_\varphi$, which is created from the automata for φ's sub-formulae by the automata operation corresponding to the top-level logical operator of φ in the following way (the automata operations preserve determinism and completeness):

(i) If $\varphi = \psi_1 \wedge \psi_2$ then, for $\mathcal{A}_{\psi_1}$ over the alphabet $\Sigma_\mathbb{X}$ and $\mathcal{A}_{\psi_2}$ over the alphabet $\Sigma_\mathbb{Y}$, the automaton $\mathcal{A}^\partial_\varphi$ over the alphabet $\Sigma_{\mathbb{X}\cup\mathbb{Y}}$ is given as $\mathcal{A}^\partial_\varphi = \pi^\flat_\mathbb{X}(\mathcal{A}_{\psi_1}) \cap \pi^\flat_\mathbb{Y}(\mathcal{A}_{\psi_2})$.

(ii) If $\varphi = \psi_1 \vee \psi_2$ then $\mathcal{A}^\partial_\varphi$ is constructed analogically to $\wedge$ (in particular, $\cup$ is used instead of $\cap$).

(iii) If $\varphi = \neg\psi$, then $\mathcal{A}^\partial_\varphi = \mathcal{A}^\complement_\psi$. The operation preserves determinism, minimality, and completeness of the transition relation, hence $\mathcal{A}_\varphi$ is taken directly as $\mathcal{A}^\partial_\varphi$, without calling the minimization.

(iv) If $\varphi = \exists X.\ \psi$, then the automaton $\mathcal{A}^\partial_\varphi$ is constructed as $det(\pi_X(\mathcal{A}_\psi) - \vec{0}^\star)$.

8.2 Formula Transformations

In this section, we describe our formula transformation algorithm. It is based on well-known rules for transformation of formulae to the *negation normal form* (NNF) and the *antiprenex form* (APF) [104], together with distributive laws. Recall that, during transformation into NNF, negations are pushed deeper into the formula so that they occur only in front of atoms, and during transformation into APF, quantifiers are pushed deeper into the formula in order to minimize their scopes (APF can be viewed as the opposite of the prenex normal form). In the following, we assume that the processed formula contains only existential quantifiers $\exists$ and Boolean connectives $\wedge, \vee$, and $\neg$.

We will discuss heuristics for choosing which formula transformation rules to apply and when to apply them so that the resulting formula is as easy as possible for the automata solver. We fine-tune the heuristics particularly for the algorithm of MONA and with respect to the specific way it processes individual logical connectives (cf. Section 8.1). Namely, MONA always works with complete DTAs (its data structures cannot even directly represent nondeterminism). Negation is implemented as complementation of such DTAs, which is cheap (it is sufficient to just invert the acceptance condition). The $\wedge$ and $\vee$ connectives are implemented through an automata product construction, which is quadratic (the

two constructions differ only in their treatment of the acceptance condition). Although the potential quadratic blow-up is not the source of the worst-case **NONELEMENTARY** complexity of WSkS, a sequence of product constructions is still exponential and in practice is often the main cause of a state explosion. Projection performed while processing the $\exists$ connective is the only operation that introduces nondeterminism, and is therefore done together with determinization, which is, in the worst case, exponential. The **NONELEMENTARY** worst-case complexity of the procedure stems from here.

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

8.2.1 Cost of Deciding a Formula

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

$$\|\varphi\| = \sum_{\psi \in sf(\varphi)} |\mathcal{A}_{\psi}| + |\mathcal{A}^{\partial}_{\psi}| \tag{8.1}$$

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

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

8.2.2 Quantifier Distribution and Scope Narrowing

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

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

Using this rule is generally beneficial for the following reasons. Disjunction is expensive, often quadratic, and the result is often significantly more complex than the arguments, even after the result's minimization. The arguments of the quantifications on the right-hand side of the rule, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\psi}$, are hence likely to be substantially smaller than the argument of quantification on the left-hand side of the rule, $\mathcal{A}_{\varphi \vee \psi}$. This is desirable since quantification

is often the most expensive operation, exponential in the worst case. Moreover, minimization often reduces the size of the result of quantification to even smaller than the size of the automaton before quantification, in which case the product construction is applied on smaller arguments after the transformation than before it. We therefore use quantifier distribution whenever applicable.

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

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

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

8.2.3 Supporting Rules

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

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

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

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

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

$$\exists \mathbb{X}.\ \varphi \wedge (\psi \vee \chi) \quad \rightsquigarrow \quad \exists \mathbb{X}.\ (\varphi \wedge \psi) \vee (\varphi \wedge \chi) \qquad (8.2)$$

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

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

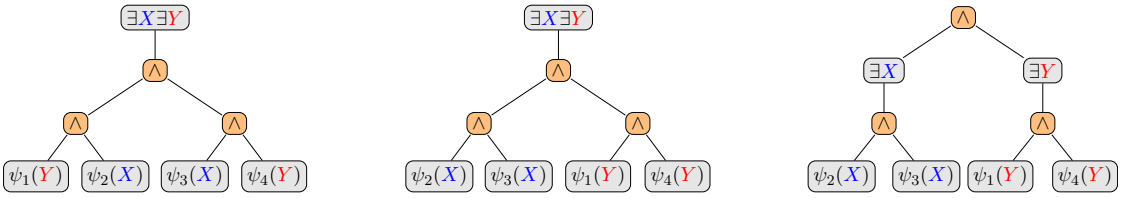


Figure 8.1: An example of how conjunction restructuring can help antiprenexing. The tree in (a) represents the formula $\chi_1 : \exists X \exists Y.\ (\psi_1 \wedge \psi_2) \wedge (\psi_3 \wedge \psi_4)$ where Y is the only free variable of ψ_1 and ψ_4 and X is the only free variable of ψ_2 and ψ_3. Notice that none of the quantifiers can be pushed inside. The tree in (b) represents a formula obtained from χ_1 by applying the associative and commutative laws to gather sub-formulae with the same free variables together, enabling the use of the (ScopeNarrow) rule. The tree in (c) is obtained from (b) by applying the rule twice.

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

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

$$\exists \mathbb{X}.\ \varphi \wedge (\psi \vee \chi) \quad \rightsquigarrow \quad \exists \mathbb{X}.\ (\varphi \wedge \psi) \vee (\varphi \wedge \chi) \quad \text{if } \|\varphi\|^{\sim} \leq \text{DISTRTHRES} \qquad (\wedge\text{-Distr})$$

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

Restructuring conjunctions. Our last transformation rule is used to facilitate quantifier scope narrowing (rule (ScopeNarrow)). It is the rule of *restructuring of conjunctions*, denoted as (Restr&Narrow). Consider a formula $\exists X_1 \ldots \exists X_m.\ \varphi$ where φ is a (possibly large and nested) conjunction. The rule can be seen as performing the following three actions: (i) reordering the sequence of quantifiers $\exists X_1 \ldots \exists X_m$ into the from most suitable for the next step, (ii) using the laws of associativity and commutativity for $\wedge$ to restructure the top-most conjunctions of φ so that the scope narrowing (w.r.t. the order induced in the previous step) can have the greatest possible effect, and (iii) performing (ScopeNarrow) to push quantifiers as deep as possible.

We will start by describing how the restructuring itself is performed. Let p be a permutation of the set $\{1, \ldots, m\}$, which induces the following reordering of the quantifiers in the transformed formula: $\exists X_{p(1)} \ldots \exists X_{p(m)}$ (we will describe how we obtain p later). Consider a sequence of quantifiers $\rho = \exists X_{p(1)} \ldots \exists X_{p(m)}$ and a formula χ. Further, let χ' be a formula obtained from $\rho.\ \chi$ by applying (ScopeNarrow) on the top-most conjunctions as long as possible. We then say that χ is *optimal for narrowing w.r.t.* ρ if no sequence of applications of the commutativity and associativity laws on the top-most conjunctions of χ' enables any more application of (ScopeNarrow). We use φ_p to denote a formula obtained by restructuring φ's top-most conjunctions using associativity and commutativity laws that is

Function `RestrAndNarrowP(`p`)`

```
1 Function Restr&Narrow(p):
2 |  Ψ := {ψ_i}_{i=1}^n;
3 |  for j := m downto 1 do
4 |  |  Φ_j := {ψ_i ∈ Ψ | X_{p(j)} ∈ fv(ψ_i)};
5 |  |  Ψ := (Ψ \ Φ_j) ∪ {∃X_{p(j)}. ⋀ Φ_j};
6 |  return (⋀ Ψ);
```

optimal for narrowing w.r.t. $\exists X_{p(1)} \dots \exists X_{p(m)}$ (there might be more such formulae; picking any of them works for us). See Figure 8.1 for an example of how $\wedge$-restructuring enables using the (ScopeNarrow) rule.

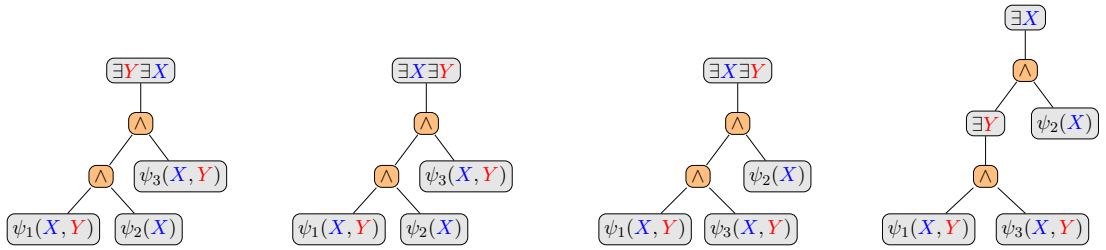
Constructing φ_p for a given permutation p is implemented as a call to the function `Restr&Narrow(`p`)`, given above. The function not only creates φ_p, but also performs, on the fly, quantifier scope narrowing (so that it is not necessary to apply (ScopeNarrow) on the result), producing a formula denoted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

Note that in the previous, we were using a permutation p of the quantifiers as a parameter of the restructuring. The way how quantifiers are ordered is important because it determines how well the restructuring can be done (see Figure 8.2 for an example).

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

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

The second heuristic works as follows. If the number of possible orderings is still too high, we split the sequence of quantifiers $\exists X_1 \dots \exists X_m$ into subsequences of the length h (except the last one which can be shorter), i.e., $\exists X_1 \dots \exists X_h; \exists X_{h+1} \dots \exists X_{2h}; \dots; \exists X_{jh} \dots \exists X_m$,



(a) **(b)** **(c)** **(d)**

Figure 8.2: An example of how reordering quantifiers can help with quantifier scope narrowing. The tree in (a) represents the formula $\chi_2 : \exists Y \exists X.\, (\psi_1(X,Y) \wedge \psi_2(X)) \wedge \psi_3(X,Y)$. Notice that for the order of quantifiers $\exists Y \exists X$, the $\wedge$-tree is optimal for narrowing. In (b), we changed the order of quantifiers to $\exists X \exists Y$. This change enables restructuring the $\wedge$-tree into a more suitable form (the tree in (c)), which, in turn, allows quantifier scope narrowing (the tree in (d)).

for some j, and try to find the best ordering for every such subsequence independently. The constant h is controlled by the parameter R&NSeqMax. The third heuristic addresses the situation when, despite the optimizations above, the application of (Restr&Narrow) may still be too costly. We therefore use the parameter R&NThres to bound the maximum size of the conjunction $\{\psi_i\}_{i=1}^{n}$ for which (Restr&Narrow) can be applied.

8.2.4 Top-level Algorithm

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

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

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

In both sequences, each iteration is started by pushing negations deeper into a formula using (PushNeg). Then, in Full, the rule (Restr&Narrow) is interleaved with (QuantDistr) to push quantifiers into conjunctions and distribute $\exists$ over $\vee$. Finally, ($\wedge$-Distr) is used to distribute $\wedge$ over $\vee$. On the other hand, in Simple, (PushNeg) is followed just by interleaving quantifier scope narrowing with distributing $\exists$ over $\vee$. (Note that it may seem that (ScopeNarrow) is only used by Simple and not by Full; in fact, the rule is used in Full internally within (Restr&Narrow).) The particular sequence of operations (Full

or SIMPLE) to be used is determined by the size of the input formula φ. In particular, if $|sf(\varphi)| \leq$ SIMPLETHRES, we pick the more expensive FULL, otherwise we pick the cheaper SIMPLE, where SIMPLETHRES is a parameter whose value specifies the threshold.

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

8.3 Automata Size Estimation

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

$$\|\varphi\|^{\sim} = \sum_{\psi \in sf(\varphi)} |\mathcal{A}_\psi|^{\sim} + |\mathcal{A}^\partial_\psi|^{\sim} \ . \tag{8.3}$$

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

$\varphi = \psi \wedge \psi' :$	$\lvert\mathcal{A}_\varphi\rvert^{\sim} = \ell_\wedge(\lvert\mathcal{A}_\psi\rvert^{\sim}, \lvert\mathcal{A}_{\psi'}\rvert^{\sim}, n)$	$\lvert\mathcal{A}^\partial_\varphi\rvert^{\sim} = \ell^\partial_\wedge(\lvert\mathcal{A}_\psi\rvert^{\sim}, \lvert\mathcal{A}_{\psi'}\rvert^{\sim}, n)$
$\varphi = \psi \vee \psi' :$	$\lvert\mathcal{A}_\varphi\rvert^{\sim} = \ell_\vee(\lvert\mathcal{A}_\psi\rvert^{\sim}, \lvert\mathcal{A}_{\psi'}\rvert^{\sim}, n)$	$\lvert\mathcal{A}^\partial_\varphi\rvert^{\sim} = \ell^\partial_\vee(\lvert\mathcal{A}_\psi\rvert^{\sim}, \lvert\mathcal{A}_{\psi'}\rvert^{\sim}, n)$
$\varphi = \exists X.\psi :$	$\lvert\mathcal{A}_\varphi\rvert^{\sim} = \ell_\exists(\lvert\mathcal{A}_\psi\rvert^{\sim})$	$\lvert\mathcal{A}^\partial_\varphi\rvert^{\sim} = \ell^\partial_\exists(\lvert\mathcal{A}_\psi\rvert^{\sim})$
$\varphi = \neg\psi :$	$\lvert\mathcal{A}_\varphi\rvert^{\sim} = \lvert\mathcal{A}_\psi\rvert^{\sim}$	$\lvert\mathcal{A}^\partial_\varphi\rvert^{\sim} = 0$
$\varphi = \psi_a :$	$\lvert\mathcal{A}_\varphi\rvert^{\sim} = \lvert\mathcal{A}_{\psi_a}\rvert$	$\lvert\mathcal{A}^\partial_\varphi\rvert^{\sim} = 0$

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

The first obvious choice for the functions ℓ and ℓ^∂ would be to use the *worst-case* size of the automata, i.e., $\ell^\partial_\star(|\mathcal{A}_\psi|^{\sim}, |\mathcal{A}_{\psi'}|^{\sim}, n) = |\mathcal{A}_\psi|^{\sim} \cdot |\mathcal{A}_{\psi'}|^{\sim}$ for $\star \in \{\wedge, \vee\}$, $\ell^\partial_\exists(|\mathcal{A}_\psi|^{\sim}) = 2^{|\mathcal{A}_\psi|^{\sim}}$, and $\ell_\bullet = \ell^\partial_\bullet$ for $\bullet \in \{\wedge, \vee, \exists\}$ (in the worst case, minimization is performed, but has no

effect). When analyzing the sizes of automata produced by MONA, we, however, noticed that the worst case happens only exceptionally, and in reality, the sizes are much smaller. In particular, the size of $\mathcal{A}^{\partial}_{\exists X.\psi}$ (resp. $\mathcal{A}_{\exists X.\psi}$) is usually linear to the size of $\mathcal{A}_{\psi}$ rather than exponential (with a few outliers where the explosion happened). Furthermore, we also noticed that there is a linear correlation between the size of $\mathcal{A}^{\partial}_{\varphi \star \psi}$ (resp. $\mathcal{A}_{\varphi \star \psi}$) and the value $\|\mathcal{A}_{\varphi}\| \cdot \|\mathcal{A}_{\psi}\|$.

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

$$\begin{aligned} \ell_\star(|\mathcal{A}_\psi|^\sim, |\mathcal{A}_{\psi'}|^\sim, n) = a^\star_n \cdot (|\mathcal{A}_\psi|^\sim \cdot |\mathcal{A}_{\psi'}|^\sim) + b^\star_n \quad \text{and} \\ \ell_\exists(|\mathcal{A}_\psi|^\sim) = a^\exists \cdot (|\mathcal{A}_\psi|^\sim) + b^\exists. \end{aligned} \tag{8.4}$$

We obtain the particular parameters $a^\exists, b^\exists$ and $a^\star_n, b^\star_n$ for every n (and their variants for ℓ^{∂}) by learning from runs of MONA. As the learning algorithm, we used *linear regression* (its particular version is discussed in Section 8.4), which is an optimization technique based on fitting input data (points in a Euclidean space) with a hyperplane such that the least square error is minimized [240]. We chose this method for its simplicity and well-predictable behavior.

8.4 Experimental Evaluation

We have implemented the antiprenexing transformations for WSkS formulae introduced in Section 8.2 as a Haskell/Python prototype tool named ANTIMONA (ANTIprenexing for MONA)[1]. The tool works as a preprocessor for MONA; it reads a file in the MONA format, applies the transformations, and produces a new file in the same format, which can then be passed to MONA. Our goal is to evaluate the impact of our optimization on MONA. Although there have recently appeared new techniques for deciding WSkS, e.g., [112, 111, 91, 273, 121], or the approach in Chapter 7, we do not focus on comparing with them, because the alternative tools are far less mature than MONA. Although they can win over MONA on limited classes of formulae, MONA performs better overall and, up to our knowledge, can still be considered the only robust and practically usable tool.

We implemented only a light-weight estimation of the costs of formulae (cf. Section 8.3). In particular, our implementation does not consider MONA's *DAGification* optimization, which first transforms a formula into a DAG where nodes corresponding to similar sub-formulae[2] are merged into one, and then constructs automata only for the nodes in the DAG (see [173] for more details). Instead, we work with the syntax tree of a formula and therefore return an over-approximation of the formula's cost estimate (some nodes are counted multiple times) .

Experimental settings. We have evaluated our technique on formulae we were able to find in the literature or obtain by personal communication (in cases where the appropriate research was not published due to problems with the scalability of MONA) and which our tool could parse. Particularly, our benchmark includes formulae from the STRAND benchmark [196], formulae from the authors of MONA [173], benchmarks for synthesis of regular specifications [135], families of parametric WS1S formulae [112], LTL formulae from [293]

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

[2] Two sub-formulae φ and φ' are *similar* if there is an order-preserving renaming of variables of φ such that after the renaming φ becomes identical to φ' [173].

Table 8.1: Parameters of the selected settings of AntiMona.

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

Table 8.2: Results of learning

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

translated to the MSO(Str) logic [89], and an experimental translation of separation logic formulae into MSO(Str) [40]. In total, our benchmark set has 103 formulae (95 WS1S and 8 WS2S).

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

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

We used the functions `lm` and `rlm` from R [3] to learn the linear estimation model. The `lm` function is a basic library function that infers a linear model using the method of least squares. On the other hand, `rlm` (from the R's `MASS` package) implements *robust fitting of linear models*, which uses a modification of the method of least squares that can deal with outliers (see the documentation of `rlm` for more information). We use the output from `rlm` whenever it is available; in some cases (e.g., when the number of data points was too small), the computation of `rlm` did not produce a result, and so we used the output of `lm` (this can happen due to the fact that `rlm` works in iterations with giving data points different weights; if the computation does not converge in a set number of iterations, the function produces no output). Moreover, we analyzed the learned models (in particular using the R^2 statistical measure [240]) and discarded those with a low fidelity—this affected models of $\ell_\wedge$ and $\ell_\vee$ with the number of shared variables n for which we did not have enough training data. The discarded models were substituted by a model for a number closest to n that had a high-fidelity. In some cases, we obtained linear models $ax + b$ with a large value of b, which caused a large bias in the computed values, especially for smaller formulae; in those cases, we modified the model by setting $b = 0$. In Table 8.2, we provide learned values of the parameters a (for ℓ_{op}) and a^∂ (for ℓ^∂_{op}) for some cases.

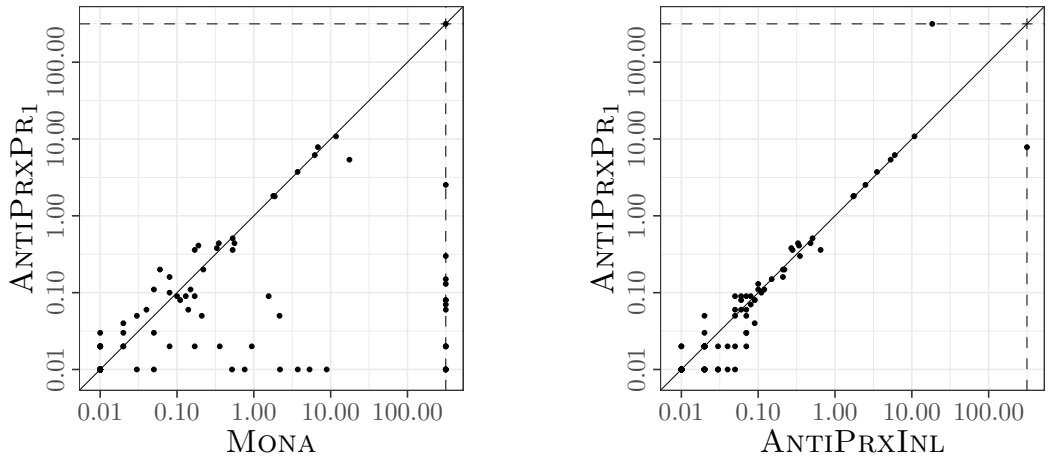


Figure 8.3: (left) A comparison of the runtime of Mona on unprocessed formulae and on formulae after AntiPrxPr$_1$, (right) a comparison of our two settings of the antiprenexing procedure. The axes are logarithmic and the dashed lines represent the cases where Mona ran out of memory or our antiprenexing did not finish (the timeout for antiprenexing was 300 s).

Parameters of antiprenexing. We experimented with different settings of parameters of our antiprenexing procedure from Section 8.2 and chose three that give the best overall performance on our benchmark set; their overview is in Section 8.1.

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

In the left-hand plot of Figure 8.3, we give a comparison of Mona and AntiPrxPr$_1$. Note that there are many cases where antiprenexing significantly shortens the time to decide a formula (the data points at the bottom of the plot) and also many formulae that can be decided only after antiprenexing (the vertical dashed line).

In the right-hand plot of Figure 8.3, we compared AntiPrxInl with AntiPrxPr$_1$. The plot shows that AntiPrxPr$_1$ more often behaves better. There are, however, two interesting cases of formulae that can only be decided by one of the settings. These are the formulae `sl` (which can be decided by AntiPrxInl in 18.43 s; AntiPrxPr$_1$ and AntiPrxPr$_2$ timeout) and `von-neumann-add` (which can be decided by AntiPrxPr$_1$ in 7.83 s, by AntiPrxPr$_2$ in 5.36 s and by Mona in 6.87 s; AntiPrxInl timeouted). The formula `sl` comes from an experimental translation of separation logic into MSO(Str) [40] (in particular of the property of the existence of a path in a symbolic heap) and the formula `von-neumann-add` encodes the fact that an 8-bit von Neumann adder is equivalent to a standard carry-chain adder [173].

In Table 8.3, we give a selection of interesting benchmarks where the first half of the table contains formulae from practical scenarios and the second half contains artificially

Table 8.3: A selection of interesting benchmarks; "–" denotes that MONA ran out of memory (OOM) or that our antiprenexing did not finish (the timeout for antiprenexing was 300 s). The column |**DAG**| gives a measure of the size of the input formula.

Formula	|**DAG**|	MONA	ANTIPRXINL	ANTIPRXPR$_1$	ANTIPRXPR$_2$	**Source**
`four-weights`	145	0.77	0.03	0.01	0.02	[135]
`smoothing`	221	17.94	5.30	5.38	0.46	[135]
`tree-weights-min`	153	12.00	10.83	10.83	10.50	[135]
`von-neumann-add`	267	6.87	–	7.83	5.36	[173]
`sl`	77	–	18.43	–	–	[40]
`lift_8.ltl0`	255	–	0.35	0.3	–	[293]
`lift_b_7.ltl0`	380	–	0.10	0.13	–	[293]
`horn_sub17`	39	2.10	0.02	0.01	0.01	[112]
`horn_sub18`	41	5.42	0.01	0.01	0.01	[112]
`horn_sub19`	43	–	0.02	0.02	0.01	[112]
Total OOM		18	1	1	5	

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

We note that our benchmark set does not contain two benchmarks that were available to us, `lift_b_8.ltl0` and `lift_b_9.ltl0` from [89], because MONA and all of the three settings of ANTIMONA presented above timeouted on them (ANTIMONA already during antiprenexing). Nonetheless, by slightly tuning the parameters of ANTIPRXINL2 (changing SIMPLETHRES to 5,000), we obtained a setting under which ANTIMONA quickly produced a formula that could be easily decided by MONA.

Discussion. The experimental results obtained from our prototype implementation show that our antiprenexing techniques can significantly reduce the time for deciding WSkS formulae—or allow the formula to be decided at all. The settings we have provided in Table 8.1 were selected for their ability to give good overall performance on the whole benchmark set, which mixes formulae of varying character. These settings are, however, not universally the best, the optimal settings for particular formulae may vary significantly. Hard formulae may be decided through tailoring the parameters to fit, as is indeed witnessed by the two last formulae mentioned above. Our parametric framework also makes it possible to fine-tune the parameters for a specific *class* of similar formulae, which typically come from specific application domains (such as verification conditions of programs of a certain kind or by translation from some given logic).

8.5 Conclusion

In this chapter, we proposed static transformations of WSkS formulae to a more suitable form allowing to speed up the automata-based decision procedure implemented in MONA. In particular, we leverage on transforming formulae into the antiprenex form. Our experimental evaluation shows that the preprocessing techniques can have a significant impact on the time for deciding formulae. As a part of our future work, we wish to automate the process of tuning the parameters of our formula transformation procedure for a given class of WSkS formulae (possibly using some machine learning approach again) and also we would like to integrate our methods directly into MONA. The content of this chapter was published in the proceedings of LPAR'20 [141].

Chapter 9

Automata in String Constraint Solving

In the previous chapters, we used automata to represent models of formulae (WSkS or Presburger arithmetic). In this chapter, we employ automata in the context of the *theory of strings (string constraints solving)*. In particular, we use automata for an efficient representation of a proof graph and proof rules. Constraint solving is a technique used as an enabling technology in many areas of formal verification and analysis, such as symbolic execution [127, 168], static analysis [288, 131], or synthesis [130, 222]. For instance, in symbolic execution, feasibility of a path in a program is tested by creating a constraint that encodes the evolution of 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 a combination of different first-order theories, such as theory of integers, reals, or strings. The theory of strings uses variables ranging over strings combined with symbol constants and concatenation (and possibly with length and/or regular constraints). Theories (or their fragments) for the integers (see, e.g., Section 6.3) and reals are well known, widely developed, and implemented in tools, while the theory of strings has started to be investigated only recently [9, 304, 45, 74, 73, 148, 191, 189, 286, 298, 7, 10, 167, 190], despite having been considered already by A. A. Markov in the late 1960s in connection with Hilbert's 10th problem [201, 102].

Although the full FO theory of strings (i.e., including quantifiers) is undecidable [235], the existential fragment is decidable (e.g., using Makanin's algorithm [197]). Most of the works dealing with string solving hence focus on interesting classes of the existential fragment and so do we in this chapter. Most current decision procedures for string constraints involve the so-called *case-split* rule. This rule performs a case split w.r.t. 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 [197], Nielsen transformation [219] (a.k.a. Levi's lemma [185]), and the procedures implemented in most state-of-the-art solvers such as Z3 [45] or CVC4 [189]. In this chapter, we will explain the general idea of our symbolic approach using Nielsen transformation, which is the simplest of the approaches; nonetheless, we believe that the approach is applicable also to other procedures.

Overview of the proposed approach. Our work brings a novel approach to an efficient symbolic handling of Nielsen transformation, a proof technique for the satisfiability checking

of word equations. Nielsen transformation is based on case-splitting rules generating a proof graph whose nodes are transformed string equations.

Consider the *word equation* $xz = yw$, where x, z, y, and w are *string variables*. When establishing satisfiability of the word equation, Nielsen transformation [219] 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 (non-disjoint) cases (i) y is a prefix of x, (ii) x is a prefix of y, (iii) x is an empty string, and (iv) y is an empty string. For these cases, Nielsen transformation generates new equations. If $xz = yw$ has a solution, then at least one of the generated equations has a solution, too. Nielsen transformation keeps applying the transformation rules on the obtained equations, building a proof graph and searching for a tautology of the form $\epsilon = \epsilon$.

During the application of Nielsen transformation, there may emerge a lot of equations that are similar and/or have common parts with other generated equations. The case split can be hence performed more efficiently if we process the common part of the proof graph together using a symbolic encoding. In this work, we use an encoding of a set of equations as a regular language, which is represented by an NFA.

Further, we show that the transformations can be encoded as *rational relations*, represented using *finite transducers*, and the whole satisfiability checking problem, including word equations with Presburger and regular constraints, can be encoded within the framework of *regular model checking* (RMC). In the past, RMC has already been considered for solving string constraints (cf. [298, 297, 300, 21]). In those approaches, the languages of the automata are, however, "models of the formula", so the approaches can be considered "model-theoretic". In our approach, the automata languages are the derived constraints. Hence the approach is closer to "proof-theoretic". We believe this novel aspect has a great potential for further investigation and can bring new ideas to the field of string constraints solving.

We implemented our approach in a prototype Python tool called Retro and evaluated its performance on two benchmark sets: $\texttt{Kepler}_{22}$ obtained from [184] and PyEx-Hard obtained by running the PyEx symbolic execution engine on Python programs [244] and collecting examples on which CVC4 or Z3 fail. Retro solved most of the problems in $\texttt{Kepler}_{22}$ (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.

Related work. The study of solving string constraint goes back to 1946, when Quine [235] showed that the first-order theory of word equations is undecidable. Makanin achieved a milestone result in [197], where he showed that the class of quantifier-free word equation is decidable. Since then, several works, e.g., [229, 230, 200, 245, 251, 119, 118, 9, 29, 190, 73, 74, 11], 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 [64].

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 [304, 304, 31, 274, 275, 10, 8, 148, 74] is to split the constraints into simpler sub-cases 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. In contrast, our approach performs case-splits symbolically.

A related topic is about *automata-based* string solvers for analyzing string-manipulating programs. ABC [21] and Stranger [297] soundly over-approximate string constraints using transducers [300]. 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.

Chapter outline. This chapter is organized as follows. Section 9.1 deals with preliminary definitions. In Section 9.2, we describe our symbolic 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, Nielsen transformation is sound and complete. In Section 9.3, we extend the technique to support *conjunctions* of *non-quadratic* word equations. In Section 9.4, we extend our approach to support arbitrary Boolean combinations of string constraints. In Section 9.5, we extend our approach also to Presburger and regular constraints. Sections 9.6 and 9.7 deal with implementation details and experimental evaluation and, finally, Section 9.8 concludes the chapter.

9.1 Preliminaries

In this section, we introduce preliminaries related to finite transducers, string constraints, MSO(Str), Nielsen transformation, and regular model checking, as they are necessary for the rest of the chapter. This section extends the definitions introduced in Chapters 2 and 6.

Words and alphabets. Let Σ be an alphabet. We define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. Given a word $w = a_1 \dots a_n$, we use $|w|_a$ to denote the number of occurrences of the character $a \in \Sigma$ in w. Further, we use $w[i]$ to denote a_i, the i-th character of w, and $w[i:]$ to denote the word $a_i \dots a_n$. When $i > n$, the value of $w[i]$ and $w[i:]$ is in both cases $\bot$, a special *undefined* value, which is different from all other values and also from itself (i.e., $\bot \neq \bot$).

Finite word transducers and relations. A *(nondeterministic) finite k-tape transducer* is a tuple $\mathcal{T} = (Q, \Sigma, \Delta, I, F)$ where 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*, $I \subseteq Q$ is a set of *initial states*, and $F \subseteq Q$ is a set of *final states*. A run π of $\mathcal{T}$ over a k-tuple of words $(w_1, \dots, w_k)$ is a sequence of transitions $(q_0, a_1^1, \dots, a_1^k, q_1),\ (q_1, a_2^1, \dots, a_2^k, q_2),\ \dots, (q_{n-1}, a_n^1, \dots, a_n^k, q_n) \in \Delta$ such that $\forall i : w_i = a_1^i a_2^i \dots a_n^i$ (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 $q_0 \in I$ and $q_n \in F$, and a k-tuple $(w_1, \dots, w_k)$ is *accepted* by $\mathcal{T}$ if there exists an accepting run of $\mathcal{T}$ over $(w_1, \dots, w_k)$. The *language* $\mathcal{L}(\mathcal{T})$ of $\mathcal{T}$ is defined as the k-ary relation $\mathcal{L}(\mathcal{T}) = \{ (w_1, \dots, w_k) \in (\Sigma^*)^k \mid (w_1, \dots, w_k) \text{ is accepted by } \mathcal{T} \}$. We call the class of relations accepted by transducers *rational relations*. $\mathcal{T}$ is *length-preserving* if no transition in Δ contains ϵ. We call the class of relations accepted by length-preserving transducers *regular relations*. Note that a *nondeterministic finite automaton* from the definition in Chapter 2 is a 1-tape finite length-preserving transducer. Given two k-ary relations R_1, R_2, we define their *concatenation* $R_1.R_2 = \{ (u_1 v_1, \dots, u_k v_k) \mid (u_1, \dots, u_k) \in R_1 \wedge (v_1, \dots, v_k) \in R_2 \}$ and given two binary relations R_1, R_2, we define their *composition* $R_1 \circ R_2 = \{ (x, z) \mid \exists y : (x, y) \in R_2 \wedge (y, z) \in R_1 \}$. 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

$$
\begin{array}{ll}
\sigma \models P \subseteq R & \text{iff } \sigma(P) \text{ is a subset of } \sigma(R) \\
\sigma \models P = R + 1 & \text{iff } \sigma(P) = \{r + 1 \mid r \in \sigma(R) \text{ and } r + 1 \leq |\sigma|\} \\
\sigma \models w[P] = a & \text{iff for all } p \in P \text{ it holds that } \sigma(w)[p] \text{ is } a \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff not } \sigma \models \varphi \\
\sigma \models \forall^{\mathbb{P}} P(\varphi) & \text{iff for all } v \subseteq \{1, \ldots, |\sigma|\} \text{ it holds that } \sigma[P \mapsto v] \models \varphi \\
\sigma \models \forall^{\mathbb{W}} w(\varphi) & \text{iff for all } v \in \Sigma^n \text{ for } n = |\sigma| \text{ it holds that } \sigma[w \mapsto v] \models \varphi
\end{array}
$$

Figure 9.1: Semantics of MSO(Str)

a language L and a binary relation R, we define the *R-image* of L as $R(L) = \{\, y \mid \exists x \in L : (x, y) \in R \,\}$.

Proposition 9.1.1 ([42])**.** *(i) The class of binary rational relations is closed under union, composition, concatenation, and iteration and is not closed under intersection and complement. (ii) For a binary rational relation R and a regular language L, the language $R(L)$ is also effectively regular (i.e., it can be computed). (iii) The class of regular relations is closed under Boolean operations.*

String constraints. Let Σ be an alphabet and $\mathbb{X}$ be a set of *string variables* ranging over Σ^* s.t. $\mathbb{X} \cap \Sigma = \emptyset$. We use $\mathbb{X}_\Sigma$ to denote the extended alphabet $\Sigma \cup \mathbb{X}$ (do not confuse with $\Sigma_{\mathbb{X}}$ defined in Section 6.2.2). An *assignment of* $\mathbb{X}$ is a mapping $I \colon \mathbb{X} \to \Sigma^*$. A *word term* is a string over the alphabet $\mathbb{X}_\Sigma$. 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 $a \in \Sigma$, $x \in \mathbb{X}_\Sigma$, and $w \in \mathbb{X}_\Sigma^*$. 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 $t \in \mathbb{X}_\Sigma^*$, a variable $x \in \mathbb{X}$, and a word term $u \in \mathbb{X}_\Sigma^*$, 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 *quadratic* if each variable has at most two occurrences, and *cubic* if each variable has at most three occurrences.

9.1.1 Monadic Second-Order Logic on Strings

In this section, we give a brief introduction to MSO(Str), first mentioned in Chapter 6 in the context of automata-based decision procedures, as it is a fundamental stone for proofs presented in the following sections. We define *monadic second-order logic on strings* over the alphabet Σ as follows. Let $\mathbb{W}$ be a countable set of *word variables* whose values range over Σ^* and $\mathbb{P}$ be a countable set of *set (second-order) position variables* whose values range over finite subsets of ω such that $\mathbb{W} \cap \mathbb{P} = \emptyset$. A formula φ of MSO(Str) is defined as

$$\varphi ::= P \subseteq R \mid P = R + 1 \mid w[P] = a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \forall^{\mathbb{P}} P(\varphi) \mid \forall^{\mathbb{W}} w(\varphi)$$

where $P, R \in \mathbb{P}$, $w \in \mathbb{W}$, and $a \in \Sigma$. 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 Figure 9.1. An *MSO(Str) variable assignment* is an assignment $\sigma \colon \mathbb{W} \cup \mathbb{P} \to (\Sigma^* \cup 2^\omega)$ 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

$$\exists^{\mathbb{P}} P(\varphi) \triangleq \neg\forall^{\mathbb{P}} P(\neg\varphi) \qquad \mathrm{Sing}(P) \triangleq \neg(P = \emptyset) \wedge \forall^{\mathbb{P}} R(R \subseteq P \to (R = \emptyset \vee R = P))$$
$$\varphi \vee \psi \triangleq \neg(\varphi \wedge \psi) \qquad p \leq r \triangleq \forall^{\mathbb{P}} T((p \in T \wedge \forall^{\mathbb{P}} u(u \in T \to \exists^{\mathbb{P}} v(v = u + 1 \wedge v \in T))) \to r \in T)$$
$$P = R \triangleq P \subseteq R \wedge R \subseteq P \qquad x = 0 \triangleq \forall^{\mathbb{P}} u(u \leq x \to u = x)$$
$$\exists^{\mathbb{W}} w(\varphi) \triangleq \neg\forall^{\mathbb{W}} w(\neg\varphi) \qquad x = \$ \triangleq \forall^{\mathbb{P}} u(x \leq u \to u = x)$$
$$p \in R \triangleq \mathrm{Sing}(p) \wedge p \subseteq R \qquad w_1[P] = w_2[R] \triangleq \bigvee_{a \in \Sigma} (w_1[P] = a \wedge w_2[R] = a)$$
$$p < r \triangleq p \leq r \wedge \neg(p = r)$$
$$P = \emptyset \triangleq \forall^{\mathbb{P}} R(P \subseteq R)$$

Figure 9.2: Syntactic sugar for MSO(Str)

$$\frac{\alpha u = \alpha v}{u = v}\ (\text{trim}) \qquad \frac{xu = v}{u[x \mapsto \epsilon] = v[x \mapsto \epsilon]}\ (x \hookrightarrow \epsilon) \qquad \frac{xu = \alpha v}{x(u[x \mapsto \alpha x]) = v[x \mapsto \alpha x]}\ (x \hookrightarrow \alpha x)$$

Figure 9.3: Rules of Nielsen transformation, for $x \in \mathbb{X}$, $\alpha \in \mathbb{X}_\Sigma$, and $u, v \in \mathbb{X}_\Sigma^*$. Symmetric rules are omitted.

omit unused variables in σ.) We use $|\sigma|$ to denote the value $|\sigma(w)|$ for any $w \in \mathbb{W}$. 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 word variables) denotes a k-ary relation over Σ^*. In particular, given an MSO(Str) string formula $\varphi(w_1, \ldots, w_k)$ with k free word variables $w_1, \ldots, w_k$, we use $\mathcal{L}(\varphi)$ to denote the relation $\{ (x_1, \ldots, x_k) \in (\Sigma^*)^k \mid \{w_1 \mapsto x_1, \ldots, w_k \mapsto x_k\} \models \varphi \}$. In the special case of $k = 1$, φ denotes a language $\mathcal{L}(\varphi) \subseteq \Sigma^*$.

Proposition 9.1.2 ([268])**.** *The class of languages denoted by MSO(Str) string formulae with* 1 *free word variable is exactly the class of regular languages. Furthermore, the class of relations denoted by MSO(Str) string formulae with k free word variables, for $k > 1$, is exactly the class of regular relations.*

Syntactic sugar for MSO(Str). In Figure 9.2, we define the standard syntactic sugar to allow us to write more concise MSO(Str) formulae. 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):

$$\forall^{\mathbb{P}} p(\varphi) \triangleq \forall^{\mathbb{P}} P(\mathrm{Sing}(P) \to \varphi[p \mapsto P])$$
$$\exists^{\mathbb{P}} p(\varphi) \triangleq \exists^{\mathbb{P}} P(\mathrm{Sing}(P) \wedge \varphi[p \mapsto P])$$

where $\varphi[p \mapsto P]$ denotes the substitution of all free occurrences of p in φ by P.

9.1.2 Nielsen Transformation

As already briefly mentioned in the introduction, Nielsen transformation can be used to check satisfiability of a conjunction of word equations. We use the three rules shown in Figure 9.3; besides the rules $x \hookrightarrow \alpha x$ and $x \hookrightarrow \epsilon$ there is also the (trim) rule, used to remove a shared prefix from both sides of the equation.

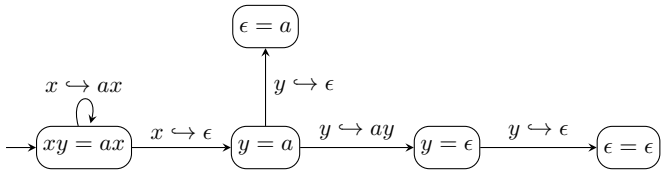


Figure 9.4: Proof graph of the equation $xy = ax$ generated by Nielsen transformation.

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. Trying all possible transformations generates a proof tree (or a graph in general) whose nodes contain conjunctions of word equations and whose edges are labelled 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 leaves in the graph is a tautology, i.e., it contains a conjunction $\epsilon = \epsilon \wedge \cdots \wedge \epsilon = \epsilon$. As an example, consider the satisfiable equation $xy = ax$ where x, y are string variables and a is a symbol with the proof graph in Figure 9.4.

Proposition 9.1.3 ([197, 95])**.** *Nielsen transformation is sound. Moreover, it is complete when the systems of word equations is quadratic.*

Proposition 9.1.3 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 $x \hookrightarrow \epsilon$ repeatedly followed by (trim).

9.1.3 Regular Model Checking

Regular model checking (RMC) [56, 6, 54] 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 triple $(\mathcal{I}, \mathcal{T}, \mathcal{D})$ 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 computation of the sequence, we can check if 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 $\bigcup_{0 \leq i < n} \mathcal{T}^i(\mathcal{I}) \supseteq \mathcal{T}^n(\mathcal{I})$, we know that we have explored all possible system configurations without reaching $\mathcal{D}$, so $\mathcal{D}$ is unreachable.

9.2 Solving Word Equations using RMC

In this section, we first focus on an inefficiency of Nielsen transformation caused by the occurrence of similar nodes in the proof graphs. Further, we describe a symbolic RMC-based framework for solving string constraints aiming at this issue. The framework is based on efficiently encoding a string constraint into a regular language and encoding steps of Nielsen transformation as a rational relation. Satisfiability of a string constraint is then reduced to a reachability problem of RMC.

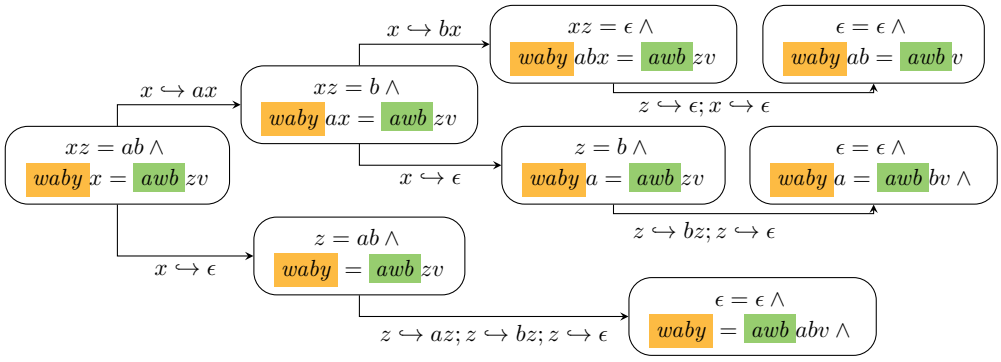


Figure 9.5: A partial proof tree of applying Nielsen transformation on $xz = ab \wedge wabyx = awbzv$. The leaves are the outcome of processing the first word equation $xz = ab$. Branches leading to contradictions are omitted.

9.2.1 Issues of Nielsen Transformation

Let us assume a conjunction of word equations solved by Nielsen transformation. Treating each of the obtained equations separately can cause some redundancy. Let us consider the example in Figure 9.5, where we apply Nielsen transformation to solve the string constraint $xz = ab \wedge wabyx = awbzv$, where v, w, x, y, and z are string variables and a and b are constant symbols. After processing the first word equation $xz = ab$, we obtain a proof tree with three similar leaf nodes $wabyab = awbv$, $wabya = awbbv$, and $waby = awbabv$, which share the prefixes $waby$ and awb on the left and right-hand side of the equations, respectively. If we continue applying Nielsen transformation on the three leaf nodes, we will create three 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 some operations to be performed multiple times. In this case, 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. To avoid these redundancies we, in the following sections, propose an efficient representation and handling of the proof graph.

9.2.2 Nielsen Transformation as Word Operations

In the following, we describe how Nielsen transformation of a single word equation can be expressed as operations on words. We view a word equation $eq : \mathtt{t}_\ell = \mathtt{t}_r$ as a pair of word terms $e_{eq} = (\mathtt{t}_\ell, \mathtt{t}_r)$ corresponding to the two sides of the equation; therefore $e_{eq} \in \mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*$. Without loss of generality we assume that $\mathtt{t}_\ell[1] \neq \mathtt{t}_r[1]$; if this is not the case, we preprocess the equation by applying the (trim) Nielsen transformation (cf. Figure 9.3) to trim the common prefix of $\mathtt{t}_\ell$ and $\mathtt{t}_r$.

Example 9.2.1. *The word equation $eq_1 : xay = yx$ is represented by the pair of word terms $e_1 = (xay, yx)$.*

A rule of Nielsen transformation (cf. Section 9.1.2) is represented using a (partial) function $\tau \colon (\mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*) \to (\mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*)$. Given a pair of word terms $(\mathtt{t}_\ell, \mathtt{t}_r)$ 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 Nielsen transformation on *eq*. Before we express the rules of Nielsen transformation, we define functions performing the corresponding substitution. For $x \in \mathbb{X}$ and $\alpha \in \mathbb{X}_\Sigma$, we define

$$\begin{aligned} \tau_{x\mapsto\alpha x} &= \{\, (\mathtt{t}_\ell, \mathtt{t}_r) \mapsto (\mathtt{t}'_\ell, \mathtt{t}'_r) \mid \mathtt{t}'_\ell = \mathtt{t}_\ell[x \mapsto \alpha x] \wedge \mathtt{t}'_r = \mathtt{t}_r[x \mapsto \alpha x] \,\} \text{ and} \\ \tau_{x\mapsto\epsilon} &= \{\, (\mathtt{t}_\ell, \mathtt{t}_r) \mapsto (\mathtt{t}'_\ell, \mathtt{t}'_r) \mid \mathtt{t}'_\ell = \mathtt{t}_\ell[x \mapsto \epsilon] \wedge \mathtt{t}'_r = \mathtt{t}_r[x \mapsto \epsilon] \,\}. \end{aligned} \tag{9.1}$$

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 9.2.2. *Consider the pair of word terms e_1 from Example 9.2.1. The application $\tau_{x\mapsto yx}(e_1)$ would produce the pair $e_2 = (yxay, yyx)$ while 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 Nielsen transformation to terminate. Let us, therefore, define the following function, which trims (the longest) matching prefix of word terms of the two sides of an equation:

$$\begin{aligned} \tau_{trim} = \{\, (\mathtt{t}_\ell, \mathtt{t}_r) \mapsto (\mathtt{t}'_\ell, \mathtt{t}'_r) \mid \exists i (\mathtt{t}_\ell[i] \neq \mathtt{t}_r[i] \wedge \forall j (j < i \rightarrow \mathtt{t}_\ell[j] = \mathtt{t}_r[j]) \\ \wedge \mathtt{t}'_\ell = \mathtt{t}_\ell[i:] \wedge \mathtt{t}'_r = \mathtt{t}_r[i:]) \,\}. \end{aligned} \tag{9.2}$$

Example 9.2.3. *Continuing in our running example, the application $\tau_{trim}(e_2)$ produces the pair $e'_2 = (xay, yx)$ while $\tau_{trim}(e_3)$ produces the pair $e'_3 = (ay, y)$.*

Now we are ready to define functions corresponding to the rules of Nielsen transformation. In particular, the rule $x \rightarrowtail \alpha x$ for $x \in \mathbb{X}$ and $\alpha \in \mathbb{X}_\Sigma$ (cf. Section 9.1.2) can be expressed using the function

$$\begin{aligned} \tau_{x \rightarrowtail \alpha x} = \tau_{trim} \circ \{\, (\mathtt{t}_\ell, \mathtt{t}_r) \mapsto \tau_{x\mapsto\alpha x}(\mathtt{t}_\ell, \mathtt{t}_r) \mid (\mathtt{t}_\ell[1] = \alpha \wedge \mathtt{t}_r[1] = x) \vee \\ (\mathtt{t}_r[1] = \alpha \wedge \mathtt{t}_\ell[1] = x) \,\} \end{aligned} \tag{9.3}$$

while the rule $x \rightarrowtail \epsilon$ for $x \in \mathbb{X}$ can be expressed as the function

$$\tau_{x \rightarrowtail \epsilon} = \tau_{trim} \circ \{\, (\mathtt{t}_\ell, \mathtt{t}_r) \mapsto \tau_{x\mapsto\epsilon}(\mathtt{t}_\ell, \mathtt{t}_r) \mid \mathtt{t}_\ell[1] = x \vee \mathtt{t}_r[1] = x \}. \tag{9.4}$$

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 graph of Nielsen transformation. In the following, we show how to perform the steps *symbolically* on a representation of a *whole set of word equations* at once.

9.2.3 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. Given a set of equations, the transformation rules are applied atomically, i.e., a single transformation rule is applied on the whole set

of equations without interleaving with other transformation rules. 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 $\alpha \in \mathbb{X}_\Sigma$. The signature of these relations is $(\mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*) \times (\mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*)$ and they are defined as follows

$$\mathcal{T}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, \alpha \in \mathbb{X}_\Sigma} \tau_{x \rightarrowtail \alpha x},$$
$$\mathcal{T}_{x \rightarrowtail \epsilon} = \bigcup_{x \in \mathbb{X}} \tau_{x \rightarrowtail \epsilon}.$$

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 $eq : \mathtt{t}_\ell = \mathtt{t}_r$, is the set $E_{eq} = \{(\mathtt{t}_\ell, \mathtt{t}_r)\}$.

Example 9.2.4. *Lifting our running example to the introduced notions over sets, we start with the set* $E_{eq} = \{e_1 = (xay, yx)\}$. *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* $S_2 = S_1 \cup \{(ax, x)\}$, *with the pair* (ax, x) *obtained from* (axy, yx) *by 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 (i) either testing whether $(\epsilon, \epsilon) \in \mathcal{T}_{step}^{rt}(E_{eq})$; this means that eq is satisfiable, or (ii) 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 9.2.5. *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* eq_1 *is unsatisfiable.*

9.2.4 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 transformation (transition) relation is a rational relation. We start by describing two possible methods of encodings that do not work and then describe the one that we use.

The first idea about how to encode a set of word equations as a regular language is to encode a pair $e_{eq} = (\mathtt{t}_\ell, \mathtt{t}_r)$ as a word $\mathtt{t}_\ell . \circledcirc . \mathtt{t}_r$, where $\circledcirc \notin \mathbb{X}_\Sigma$. 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).

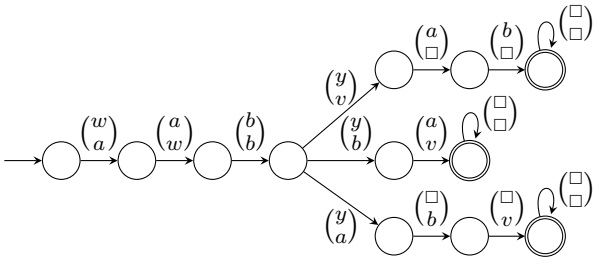


Figure 9.6: A finite automaton encoding the three equations $wabyab = awbv$, $wabya = awbbv$, and $waby = awbabv$.

Another attempt of an encoding may be encoding $e_{eq} = (\mathsf{t}_\ell, \mathsf{t}_r)$ as a rational binary relation, represented, e.g., by a (non-length-preserving) 2-tape transducer (with a tape for t_ℓ and a tape for t_r) and use 4-tape transducers to represent the transformations (with two input tapes for t_ℓ, t_r and two output tapes for t'_ℓ, t'_r). The transducers implementing $\tau_{x \mapsto yx}$ and $\tau_{x \mapsto \epsilon}$ can be constructed easily and so can be the transducer implementing τ_{trim}, so this solution looks appealing. One, however, quickly realizes an issue with computing $\mathcal{T}_{step}(E_{eq})$. In particular, since E_{eq} and $\mathcal{T}_{step}$ are both represented as rational relations, the intersection $(E_{eq} \times \mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*) \cap \mathcal{T}_{step}$, which needs to be computed first, may not be rational. Why? Consider $E_{eq} = \{\, (a^m b^n, c^m) \mid m, n \geq 0 \,\}$ and $\mathcal{T}_{step} = \{\, (a^m b^n, c^n, \epsilon, \epsilon) \mid m, n \geq 0 \,\}$. The intersection $(E_{eq} \times \mathbb{X}_\Sigma^* \times \mathbb{X}_\Sigma^*) \cap \mathcal{T}_{step} = \{\, (a^n b^n, c^n, \epsilon, \epsilon) \mid n \geq 0 \,\}$ is clearly not rational.

9.2.5 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 $\mathbb{X}^2_{\Sigma,\square}$, where $\mathbb{X}_{\Sigma,\square} = \mathbb{X}_\Sigma \cup \{\square\}$, using an NFA. For instance, $e_1 = (xay, yx)$ would be represented by the regular language $\binom{x}{y}\binom{a}{x}\binom{y}{\square}\binom{\square}{\square}^*$. Formally, we first define the *equation encoding function* $\mathsf{eqencode}\colon (\mathbb{X}_\Sigma^*)^2 \to (\mathbb{X}^2_{\Sigma,\square})^*$ such that for $\mathsf{t}_\ell = a_1 \dots a_n$ and $\mathsf{t}_r = b_1 \dots b_m$ (without loss of generality we assume that $|\mathsf{t}_\ell| \geq |\mathsf{t}_r|$), we have $\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r) = \binom{a_1}{b_1}\binom{a_2}{b_2} \dots \binom{a_m}{b_m}\binom{a_{m+1}}{\square} \dots \binom{a_n}{\square}$. We lift $\mathsf{eqencode}$ to sets in the usual way and to relations on pairs of word terms τ as $\mathsf{eqencode}(\tau) = \{\, (\mathsf{eqencode}(\mathsf{t}_\ell, \mathsf{t}_r), \mathsf{eqencode}(\mathsf{t}'_\ell, \mathsf{t}'_r)) \mid ((\mathsf{t}_\ell, \mathsf{t}_r), (\mathsf{t}'_\ell, \mathsf{t}'_r)) \in \tau \,\}$.

Let σ be a symbol. We define the *padding* of a k-tuple of words $(w_1, \dots, w_k)$ with respect to σ as the set $\mathsf{pad}_\sigma(w_1, \dots, w_k) = \{(w'_1, \dots, w'_k) \mid w'_i \in w_i.\{\sigma\}^*\}$, i.e., it is a set of k-tuples obtained from $(w_1, \dots, w_k)$ by extending some of the words by an arbitrary number of σ's. We lift pad_σ to a k-ary relation R as $\mathsf{pad}_\sigma(R) = \bigcup_{x \in R} \mathsf{pad}_\sigma(x)$. Finally, we define the function encode, which we use for encoding word equations into regular languages and word operations into rational relations, as $\mathsf{encode} = \mathsf{pad}_{\binom{\square}{\square}} \circ \mathsf{eqencode}$.

Example 9.2.6. *Consider the equations $wabyab = awbv$, $wabya = awbbv$, and $waby = awbabv$ considered in Section 9.2.1. These equations are encoded into the regular language represented by an NFA in Figure 9.6.*

Properties of encode are given by the following lemmas.

Lemma 9.2.1. *If T is a binary regular relation on pairs of word terms, then $\mathsf{encode}(T)$ is rational. If T is a unary regular relation on pairs of word terms, then $\mathsf{encode}(T)$ is regular.*

Proof. We show an idea how to construct a transducer $\mathcal{T}'$ for $\mathsf{eqencode}(T)$. Since T is a regular relation, we can modify the transducer recognizing T to obtain a transducer $\mathcal{T}'$ recognizing $\mathsf{eqencode}(T)$ (by a modification of the transition relation and handling of the padding symbol $\square$).

If T is unary, we directly have that $\mathsf{encode}(T) = \mathcal{L}(\mathcal{T}').\binom{\square}{\square}^*$, which is a regular language. Further, let us assume that T is a binary relation on pairs of word terms. In the next step, consider the relations $\tau_{+\mathsf{pad}} = \left\{(w, w') \mid w \in (\mathbb{X}^2_{\Sigma,\square})^*, w' \in w.\binom{\square}{\square}^*\right\}$ and $\tau_{-\mathsf{pad}} = \left\{(w, w') \mid w' \in (\mathbb{X}^2_{\Sigma,\square})^*, w \in w'.\binom{\square}{\square}^*\right\}$ adding and removing padding, respectively. These relations are rational. Then, observe that $\mathsf{encode}(T) = \tau_{+\mathsf{pad}} \circ \mathcal{L}(\mathcal{T}') \circ \tau_{-\mathsf{pad}}$. Finally, from Proposition 9.1.1, we obtain the rationality of $\mathsf{encode}(T)$. □

Lemma 9.2.2. *Given a word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ for $\mathsf{t}_\ell, \mathsf{t}_\ell \in \mathbb{X}^*_\Sigma$, the set $\mathsf{encode}(eq)$ is regular.*

Proof. Without loss of generality we assume that $|\mathsf{t}_\ell| \leq |\mathsf{t}_r|$. We give the following MSO(Str) formula that encodes eq:

$$\begin{aligned}\varphi_{eq}(w, w') \triangleq \bigwedge_{1 \leq k \leq |\mathsf{t}_\ell|} w[k] = \mathsf{t}_\ell[k] \;\wedge \bigwedge_{|\mathsf{t}_\ell| < k \leq |\mathsf{t}_r|} w[k] = \square \;\wedge \\ \bigwedge_{1 \leq k \leq |\mathsf{t}_r|} w'[k] = \mathsf{t}_r[k] \wedge \\ \forall^{\mathbb{P}} p((p > |\mathsf{t}_r|) \rightarrow (w[p] = \square \wedge w'[p] = \square))\end{aligned} \tag{9.5}$$

From Proposition 9.1.2, it follows that $\mathcal{L}(\varphi_{eq})$ is regular. Since $\mathcal{L}(\varphi_{eq})$ is a unary relation on pairs of word terms, from Lemma 9.2.1 we have that $\mathsf{encode}(\mathcal{L}(\varphi_{eq}))$ is regular. □

Observe that because of the padding part, which introduces an unbounded number of padding symbols at the end of an encoded relation, even if T is finite, $\mathsf{encode}(T)$ is infinite. 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 $L = \binom{x}{y}^*$, 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 $\{\, \mathsf{encode}(u, v) \mid \exists n : u = (yx)^n \wedge v = y^n \square^n \,\}$ 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 when the equation eq is quadratic, and further extend our algorithm to equations with more occurrences of variables in Section 9.3. Let us define the following, more general, restriction of $\tau_{x \rightarrowtail \alpha x}$ to equations with at most $i \in \omega$ occurrences of variable x as $\tau^{\leq i}_{x \rightarrowtail \alpha x} = \tau_{x \rightarrowtail \alpha x} \cap \{\, ((\mathsf{t}_\ell, \mathsf{t}_r), (w, w')) \mid w, w' \in \mathbb{X}^*_\Sigma, |\mathsf{t}_\ell.\mathsf{t}_r|_x \leq i \,\}$. We define $\tau^{\leq i}_{x \rightarrowtail \epsilon}$, $\tau^{\leq i}_{x \mapsto \alpha x}$, and $\tau^{\leq i}_{x \mapsto \epsilon}$ similarly.

Lemma 9.2.3. *Given $i \in \omega$, the relations $\mathsf{encode}\left(\tau^{\leq i}_{x \rightarrowtail \alpha x}\right)$ and $\mathsf{encode}\left(\tau^{\leq i}_{x \rightarrowtail \epsilon}\right)$ are rational.*

Proof. We begin with a definition of some useful predicates:

$$ordered(k_1, \ldots, k_m) \triangleq \bigwedge_{1 \leq i < m} k_i < k_{i+1} \tag{9.6}$$

$$alleq^w_x(k_1, \ldots, k_m) \triangleq \bigwedge_{1 \leq i \leq m} w[k_i] = x \tag{9.7}$$

We use the following MSO(Str) formula to define the transformation $x \rightarrowtail \epsilon$ for n occurrences of x in a single string.

$$\begin{aligned}\psi^n_{x\rightarrowtail\epsilon}(w,w') \triangleq \exists i_1,\ldots,i_n\big(&ordered(i_1,\ldots,i_n) \wedge alleq^w_x(i_1,\ldots,i_n) \wedge \\ &\forall j(j<i_1 \rightarrow w'[j]=w[j]) \wedge \\ &\bigwedge_{1\le k<n} \forall j((i_k<j<i_{k+1}) \rightarrow w'[j-k]=w[j]) \wedge \\ &\forall j(i_n<j \rightarrow w'[j-n]=w[j]) \wedge \\ &\bigwedge_{1\le k\le n} w'[\$-k]=\square\big)\end{aligned} \tag{9.8}$$

We extend it to describe the relation on pairs of strings:

$$\begin{aligned}\varphi^n_{x\rightarrowtail\epsilon}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \triangleq\ &(\mathsf{t}_\ell[0]=x \vee \mathsf{t}_r[0]=x) \\ &\wedge \bigvee_{0\le k\le n} \psi^k_{x\rightarrowtail\epsilon}(\mathsf{t}_\ell,\mathsf{t}'_\ell) \wedge \psi^{n-k}_{x\rightarrowtail\epsilon}(\mathsf{t}_r,\mathsf{t}'_r)\end{aligned} \tag{9.9}$$

$$\varphi^{\le n}_{x\rightarrowtail\epsilon}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \triangleq \bigvee_{0\le k\le n} \varphi^k_{x\rightarrowtail\epsilon}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \tag{9.10}$$

Next, we define the transformation $x \rightarrowtail \alpha x$ for n occurrences of x on a single string.

$$\begin{aligned}\psi^n_{x\rightarrowtail\alpha x}(w,w') \triangleq \exists i_1,\ldots,i_n\big(&ordered(i_1,\ldots,i_n) \wedge alleq^w_x(i_1,\ldots,i_n) \wedge \\ &\forall j(j\le i_1 \rightarrow w'[j]=w[j]) \wedge \\ &\bigwedge_{1\le k<n} w'[i_k+k]=\alpha \wedge \forall j((i_k<j\le i_{k+1}) \rightarrow w'[j+k]=w[j]) \wedge \\ &w'[i_n+n]=\alpha \wedge \forall j(i_n<j \rightarrow w'[j-n]=w[j]) \wedge \\ &\bigwedge_{1\le k\le n} w[\$-k]=\square\big)\end{aligned} \tag{9.11}$$

We extend it to describe the relation on pairs of strings:

$$\begin{aligned}\varphi^n_{x\rightarrowtail\alpha x}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \triangleq\ &(\mathsf{t}_\ell[0]=x \wedge \mathsf{t}_r[0]=\alpha) \\ &\vee (\mathsf{t}_r[0]=x \wedge \mathsf{t}_\ell[0]=\alpha) \\ &\wedge \bigvee_{0\le k\le n} \psi^k_{x\rightarrowtail\alpha x}(\mathsf{t}_\ell,\mathsf{t}'_\ell) \wedge \psi^{n-k}_{x\rightarrowtail\alpha x}(\mathsf{t}_r,\mathsf{t}'_r)\end{aligned} \tag{9.12}$$

$$\varphi^{\le n}_{x\rightarrowtail\alpha x}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \triangleq \bigvee_{0\le k\le n} \varphi^k_{x\rightarrowtail\alpha x}(\mathsf{t}_\ell,\mathsf{t}_r,\mathsf{t}'_\ell,\mathsf{t}'_r) \tag{9.13}$$

Let us consider a relation $\tau^{\mathsf{eqen}}_{trim} = \mathsf{eqencode} \circ \tau_{trim}$ removing common prefix of encoded equations. This relation can be implemented using a simple transducer replacing from the prefix symbols of the form $\binom{x}{x}$ with ϵ. We also consider relations $\tau_{+\mathsf{pad}} = \left\{(w,w') \mid w \in (\mathbb{X}^2_{\Sigma,\square})^*, w' \in w.\binom{\square}{\square}^*\right\}$ and $\tau_{-\mathsf{pad}} = \left\{(w,w') \mid w' \in (\mathbb{X}^2_{\Sigma,\square})^*, w \in w'.\binom{\square}{\square}^*\right\}$ adding and removing padding, respectively. These relations are rational. We need them because $\mathcal{L}\left(\varphi^{\le n}_{x\rightarrowtail\alpha x}\right)$ is length-preserving and hence we need to remove padding to obtain the "unpadded" pairs. Now observe that $\mathsf{encode}\left(\tau^{\le n}_{x\rightarrowtail\alpha x}\right) = \tau^{\mathsf{eqen}}_{trim} \circ \tau_{+\mathsf{pad}} \circ \tau_{-\mathsf{pad}} \circ \mathsf{encode}\left(\mathcal{L}\left(\varphi^{\le n}_{x\rightarrowtail\alpha x}\right)\right) \circ \tau_{+\mathsf{pad}}$. Finally, from Proposition 9.1.1 and Lemma 9.2.1, we have that $\mathsf{encode}\left(\tau^{\le n}_{x\rightarrowtail\alpha x}\right)$ is rational. Similarly, we can show that $\mathsf{encode}\left(\tau^{\le n}_{x\rightarrowtail\epsilon}\right)$ is rational. □

Algorithm 3: Solving a string constraint φ using RMC

Input: Encoding $\mathcal{I}$ of a formula φ (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 φ if φ is satisfiable, *false* otherwise

1 $reach_0 := \emptyset$;
2 $reach_1 := \mathcal{I}$;
3 $processed := reach_0$;
4 $\mathcal{T} := \mathcal{T}_{x \rightarrowtail \alpha x} \cup \mathcal{T}_{x \rightarrowtail \epsilon}$;
5 $i := 1$;
6 **while** $reach_i \not\subseteq processed$ **do**
7 **if** $\mathcal{D} \cap reach_i \neq \emptyset$ **then**
8 **return** $ExtractModel(reach_1, \ldots, reach_i)$;
9 $processed := processed \cup reach_i$;
10 $reach_{i+1} := \mathcal{T}(reach_i)$;
11 $i{+}{+}$;
12 **return** *false*;

$$\mathcal{I}^{eq} = \mathsf{encode}\,(\mathsf{t}_\ell, \mathsf{t}_r) \qquad \mathcal{D}^{eq} = \left\{ \binom{\square}{\square} \right\}^*$$

$$\mathcal{T}^{eq}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, \alpha \in \mathbb{X}_\Sigma} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \alpha x}\right) \qquad \mathcal{T}^{eq}_{x \rightarrowtail \epsilon} = \bigcup_{x \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \epsilon}\right)$$

Figure 9.7: RMC instantiation for a quadratic equation

In Algorithm 3, we give a high-level algorithm for solving string constraints using RMC. The algorithm is parameterized by the following inputs: a regular language $\mathcal{I}$ encoding a formula φ (the initial set), rational relations $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \epsilon}$, and 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 transition relation $\mathcal{T}$ computed in Line 4, looking for an element w_i from $\mathcal{D}$. If such an element is found in $reach_i$, we extract a model of the original word equation by starting a backward run from w_i, computing pre-images $w_{i-1}, \ldots, w_1$ 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 transformation that was performed.

Our first instantiation of the algorithm is for checking satisfiability of a single quadratic word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$. We instantiate the RMC problem with $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{eq})$ defined in Figure 9.7.

Lemma 9.2.4. *The relations $\mathcal{T}^{eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{eq}_{x \rightarrowtail \epsilon}$ are rational.*

Proof. See that $\mathcal{T}^{eq}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, \alpha \in \mathbb{X}_\Sigma} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \alpha x}\right)$ and $\mathcal{T}^{eq}_{x \rightarrowtail \epsilon} = \bigcup_{x \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \epsilon}\right)$. From Proposition 9.1.1 and Lemma 9.2.3, we have that $\mathcal{T}^{eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{eq}_{x \rightarrowtail \epsilon}$ are rational. □

Lemma 9.2.5. *If $eq : \mathsf{t}_\ell = \mathsf{t}_r$ is quadratic then Section 3 instantiated with $(\mathcal{I}^{eq}, \mathcal{T}^{eq}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{eq}_{x \rightarrowtail \epsilon}, \mathcal{D}^{eq})$ is sound and complete.*

Proof. (Sketch) We encode the nodes of a Nielsen proof graph as strings. The initial node $\mathsf{t}_\ell = \mathsf{t}_r$ corresponds to $\mathcal{I}^{eq}$. Since we use padding, the final node $\epsilon = \epsilon$ corresponds to

the set $\mathcal{D}^{eq}$. The relations $\mathcal{T}^{eq}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{eq}_{x \rightarrowtail \epsilon}$ encode the Nielsen rules $x \rightarrowtail \alpha x$ and $x \rightarrowtail \epsilon$. Soundness and completeness then follows from Proposition 9.1.3. □

9.3 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 $\Phi : \mathsf{t}^1_\ell = \mathsf{t}^1_r \wedge \mathsf{t}^2_\ell = \mathsf{t}^2_r \wedge \ldots \wedge \mathsf{t}^n_\ell = \mathsf{t}^n_r$. 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

$$\mathsf{encode}(\Phi) = \mathsf{encode}(\mathsf{t}^1_\ell, \mathsf{t}^1_r).\left\{\binom{\#}{\#}\right\}.\ldots.\left\{\binom{\#}{\#}\right\}.\mathsf{encode}(\mathsf{t}^n_\ell, \mathsf{t}^n_r), \tag{9.14}$$

where # is a delimiter symbol, $\# \notin \mathbb{X}_\Sigma$. From Lemma 9.2.2 we know that $\mathsf{encode}(\mathsf{t}^i_\ell, \mathsf{t}^i_r)$ is regular for all $1 \leq i \leq n$. Moreover, since regular languages are closed under concatenation (Proposition 9.1.1), 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 $\left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$.

For the transition relation, we need to extend $\tau^{\leq i}_{x \rightarrowtail \alpha x}$ and $\tau^{\leq i}_{x \rightarrowtail \epsilon}$ from Section 9.2 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 on 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 $\left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$. 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:

$$T^{eqs,i}_{x \rightarrowtail \alpha x} = T_{skip}.\mathsf{encode}\left(\tau^{\leq i}_{x \rightarrowtail \alpha x}\right).\left(\left\{\binom{\#}{\#} \mapsto \binom{\#}{\#}\right\}.\mathsf{encode}\left(\tau_{trim} \circ \tau^{\leq i}_{x \mapsto \alpha x}\right)\right)^*, \tag{9.15}$$

where $T_{skip} = \left\{\binom{\square}{\square} \mapsto \binom{\square}{\square}, \binom{\#}{\#} \mapsto \binom{\#}{\#}\right\}^*$. The relation $T^{eqs,i}_{x \rightarrowtail \epsilon}$ is defined similarly.

Lemma 9.3.1. *The relations $T^{eqs,i}_{x \rightarrowtail \alpha x}$ and $T^{eqs,i}_{x \rightarrowtail \epsilon}$ are rational.*

Proof. We prove only rationality of $T^{eqs,i}_{x \rightarrowtail \alpha x}$; rationality of $T^{eqs,i}_{x \rightarrowtail \epsilon}$ can be proved analogously. We can prove $\mathsf{encode}\left(\tau_{trim} \circ \tau^{\leq i}_{x \mapsto \alpha x}\right)$ is rational using a similar proof to Lemma 9.2.3, where we proved that $\mathsf{encode}\left(\tau^{\leq i}_{x \rightarrowtail \alpha x}\right)$ is rational. Further, it is easy to see that T_{skip} is rational. The lemma then follows from Proposition 9.1.1 (rational relations are closed under concatenation and iteration). □

9.3.1 Quadratic Case

When the system Φ is quadratic, its satisfiability problem can be reduced to an RMC problem $(\mathcal{I}^{q\text{-}eqs}_\Phi, \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \epsilon}, \mathcal{D}^{q\text{-}eqs})$ where the items are defined in Figure 9.8. Rationality of $\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{q\text{-}eqs}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 9.1.1. The soundness and completeness of our procedure for a system of quadratic word equations is summarized by the following lemma.

$$\mathcal{I}_{\Phi}^{q\text{-}eqs} = \mathsf{encode}(\Phi) \qquad \mathcal{D}^{q\text{-}eqs} = \left\{\binom{\square}{\square}, \binom{\#}{\#}\right\}^*$$
$$\mathcal{T}_{x \rightarrowtail \alpha x}^{q\text{-}eqs} = \bigcup_{x \in \mathbb{X}, \alpha \in \mathbb{X}_\Sigma} T_{x \rightarrowtail \alpha x}^{eqs,2} \qquad \mathcal{T}_{x \rightarrowtail \epsilon}^{q\text{-}eqs} = \bigcup_{x \in \mathbb{X}} T_{x \rightarrowtail \epsilon}^{eqs,2}$$

Figure 9.8: RMC instantiation for a system of quadratic equations

Algorithm 4: Transformation to a cubic system of equations

Input: System of word equations Φ
Output: Equisatisfiable cubic system of word equations Ψ

```
Ψ := Φ;
while There is a word variable x that occurs more than three times in Ψ do
    Replace two occurrences of x in Φ by a fresh string variable x' to obtain a new
     system Ψ';
    Ψ := Ψ' ∧ x = x';
return Ψ;
```

Lemma 9.3.2. *If Φ is quadratic then Algorithm 3 instantiated with $(\mathcal{I}_{\Phi}^{q\text{-}eqs}, \mathcal{T}_{x \rightarrowtail \alpha x}^{q\text{-}eqs} \cup \mathcal{T}_{x \rightarrowtail \epsilon}^{q\text{-}eqs}, \mathcal{D}^{q\text{-}eqs})$ is sound and complete.*

Proof. (Sketch) We can see nodes of the Nielsen proof graph as sets of word equations. The set of equations denotes conjunction of equations in the set (see Section 9.1.2). The transformation rules are generalized to take into account the set of equations. For instance consider a rule $x \rightarrowtail \alpha x$ applied on a set S. Then, $x \rightarrowtail \alpha x$ is applied on an equation $eq \in S$ (if possible) and to the remaining equations in S the substitution $x \mapsto \alpha x$ is applied. The initial node is a set corresponding to a system Φ and the final node is $\{\epsilon = \epsilon\}$. The soundness and completeness is not affected by the choice of equation from a set. Therefore we can consider ordered sets and use the first equation for a transformation.

The soundness and correctness of our algorithm follows from the fact that the initial node corresponds to $\mathcal{I}_{\Phi}^{q\text{-}eqs}$, the final node corresponding to language $\mathcal{D}^{q\text{-}eqs}$ (the delimiters are not removed from strings) and the transformations $x \rightarrowtail \alpha x$ and $x \rightarrowtail \epsilon$ correspond to $\mathcal{T}_{x \rightarrowtail \alpha x}^{q\text{-}eqs}$ and $\mathcal{T}_{x \rightarrowtail \epsilon}^{q\text{-}eqs}$, respectively. □

9.3.2 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 4 to a general system of string constraints Φ to get an equisatisfiable cubic system of word equations Φ'. Then, we can use the transition relations $T_{x \rightarrowtail \alpha x}^{eqs,3}$ and $T_{x \rightarrowtail \epsilon}^{eqs,3}$ to construct transformations of the encoded system Φ'.

Lemma 9.3.3. *Any system of word equations can be transformed by Algorithm 4 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 4, the number of occurrences of a variable x is decreased by one and a new variable x' with three occurrences is introduced. □

$$\mathcal{I}_{\Phi}^{eqs} = \mathsf{encode}(\Phi') \qquad \mathcal{D}^{eqs} = \left\{ \binom{\square}{\square}, \binom{\#}{\#} \right\}^*$$

$$\mathcal{T}_{x \rightarrowtail \alpha x}^{v_i, eqs} = T_{\mathcal{C}_{v_i}} \circ \bigcup_{x \in \mathbb{X}, \alpha \in \mathbb{X}_\Sigma} T_{x \rightarrowtail \alpha x}^{eqs,3} \qquad \mathcal{T}_{x \rightarrowtail \epsilon}^{v_i, eqs} = T_{\mathcal{C}_{v_i}} \circ \bigcup_{x \in \mathbb{X}} T_{x \rightarrowtail \epsilon}^{eqs,3}$$

Figure 9.9: RMC instantiation for a system of cubic equations

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 $x.\mathtt{t}_\ell = y.\mathtt{t}_r \wedge \Phi$, where x and y are string variables and $\mathtt{t}_\ell$ and $\mathtt{t}_r$ are word terms. If we apply the transformation $x \rightarrowtail yx$, we will obtain $x(\mathtt{t}_\ell[x \mapsto yx]) = \mathtt{t}_r[x \mapsto yx] \wedge \Phi[x \mapsto yx]$. Observe that (i) the number of occurrences of y is first *reduced by one* because the first y on the right-hand side of $x.\mathtt{t}_\ell = y.\mathtt{t}_r$ is removed and (ii) then the number of occurrences of y can be at most *increased by two* because there exist at most two occurrences of x in $\mathtt{t}_\ell$, $\mathtt{t}_r$, and Φ. Therefore, after the transformation $x \rightarrowtail yx$, a cubic system of word equations might become *(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 9.3.4. *The relation $T_{\mathcal{C}_v}$ performing the transformation $\mathcal{C}_v$ on an encoded single-quartic system of equations is rational.*

Proof. (Sketch) 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 $\mathcal{T}_{x,x_i}^{sq}$ 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 (only need to count up to four). For an encoding of a x-quartic system of word equations, the transducer $\mathcal{T}_{x,x_i}^{sq}$ returns an encoding that is obtained by replacing first two occurrences of x from the input to x_i and at the end appending language $\binom{\#}{\#}\binom{x}{x_i}\binom{\square}{\square}^*$.

In the second step, we create the transducer $\mathcal{T}_{cub}$ that accepts only encodings of a cubic system of word equation and returns the same encodings. Now we have

$$T_{\mathcal{C}_v} = \mathcal{L}\left(\mathcal{T}_{cub}\right) \cup \bigcup_{x \in \mathbb{X}} \mathcal{L}\left(\mathcal{T}_{x,v}^{sq}\right). \tag{9.16}$$

The lemma then follows by Proposition 9.1.1. □

To express solving a system of string constraints Φ in the terms of a (modified) RMC, we first convert Φ (using Algorithm 4) to an equisatisfiable cubic system Φ'. The satisfiability of a system of word equations Φ can be reduced to a modified RMC problem $(\mathcal{I}_{\Phi}^{eqs}, \mathcal{T}_{x \rightarrowtail \alpha x}^{v_i, eqs} \cup \mathcal{T}_{x \rightarrowtail \epsilon}^{v_i, eqs}, \mathcal{D}^{eqs})$ instantiating Algorithm 3 with components given in Figure 9.9.

For the modified RMC algorithm, we need to assume $v_i \notin \mathbb{X}_\Sigma$. We also need to update Line 4 of Algorithm 3 to $\mathcal{T}^{v_i} := \mathcal{T}_{x \rightarrowtail \alpha x}^{v_i} \cup \mathcal{T}_{x \rightarrowtail \epsilon}^{v_i}$ and Line 10 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. Rationality of $\mathcal{T}^{v_i,eqs}_{x \mapsto \alpha x}$ and $\mathcal{T}^{v_i,eqs}_{x \mapsto \epsilon}$ follows directly from Proposition 9.1.1.

Lemma 9.3.5. *The modified Algorithm 3 instantiated with* $(\mathcal{I}^{eqs}_{\Phi}, \mathcal{T}^{v_i,eqs}_{x \mapsto \alpha x} \cup \mathcal{T}^{v_i,eqs}_{x \mapsto \epsilon}, \mathcal{D}^{eqs})$ *is sound if* Φ *is cubic.*

Proof. (Sketch) Consider the generalized proof graph from the proof of Lemma 9.3.2. For an arbitrary system of word equations this graph may be infinite. However, since our algorithm implements BFS strategy, our algorithm is sound in proving Φ is satisfiable. □

Completeness. Since 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 [197], is our future work.

9.4 Handling a 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. The negation of word equations can be handled in the standard way. For instance, we can use the approach in [9] to convert a negated word equation $\mathsf{t}_\ell \neq \mathsf{t}_r$ to the string constraint

$$\bigvee_{c \in \Sigma} (\mathsf{t}_\ell = \mathsf{t}_r.cx \vee \mathsf{t}_\ell.cx = \mathsf{t}_r) \qquad \vee \bigvee_{c_1,c_2 \in \Sigma, c_1 \neq c_2} (\mathsf{t}_\ell = x_3 c_1 x_1 \wedge \mathsf{t}_r = x_3 c_2 x_2). \tag{9.17}$$

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 x_3 and start to differ in the next symbols 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 to the size of the original formula. Note that we cannot use the Tseytin transformation as it may introduce new negations.

Let us now focus on how to express solving a string constraint Φ composed of arbitrary Boolean combination of word equations using a (modified) RMC. We start by removing inequalities in Φ using (9.17), then we convert the system without inequalities into CNF, and, finally, apply the Algorithm 4 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 Section 9.3.2. 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 $\mathsf{encode}(c) = \bigcup_{1 \leq j \leq n} \mathsf{encode}\left(\mathsf{t}^j_\ell, \mathsf{t}^j_r\right)$. Then, the initial configuration for Φ' is given as

$$\mathcal{I}^{sc}_{\Phi'} = \mathsf{encode}(c_1).\left\{\binom{\#}{\#}\right\}.\ldots.\left\{\binom{\#}{\#}\right\}.\mathsf{encode}(c_m), \tag{9.18}$$

where Φ' is of the form $\Phi' : c_1 \wedge \ldots \wedge c_m$ and each clause c_i is of the form $c_i = (\mathsf{t}^1_\ell = \mathsf{t}^1_r \vee \ldots \vee \mathsf{t}^{n_i}_\ell = \mathsf{t}^{n_i}_r)$. We obtain the following lemma directly from Proposition 9.1.1.

Lemma 9.4.1. *The initial set* $\mathcal{I}^{sc}_{\Phi'}$ *is regular.*

The transition relation and the destination set are the same as the ones in the previous section, i.e., $\mathcal{T}^{v_i,sc}_{x \mapsto \alpha x} = \mathcal{T}^{v_i,eqs}_{x \mapsto \alpha x}$, $\mathcal{T}^{v_i,sc}_{x \mapsto \epsilon} = \mathcal{T}^{v_i,eqs}_{x \mapsto \epsilon}$, and $\mathcal{D}^{sc} = \mathcal{D}^{eqs}$. The soundness of our algorithm for a Boolean combination of word equations is summarized by the following lemma.

Lemma 9.4.2. *Given a Boolean combination of word equations Φ, modified Algorithm 3 instantiated with $(\mathcal{I}^{sc}_{\Phi'}, \mathcal{T}^{v_i,sc}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{v_i,sc}_{x \rightarrowtail \epsilon}, \mathcal{D}^{sc})$ is sound.*

Proof. (Sketch) A system of full word equations can be converted according to steps described above to an equisatisfiable system in CNF $\Psi : \bigwedge_{i=1}^{n} c_i$ where c_i contains only equalities. Then, Ψ is satisfiable if there is some $\phi : \bigwedge_{i=1}^{n} \mathsf{t}^i_\ell = \mathsf{t}^i_r$ where $\mathsf{t}^i_\ell = \mathsf{t}^i_r \in c_i$. Moreover, we have $\mathsf{encode}(\phi) \in \mathcal{I}^{sc}_{\Phi}$. From Lemma 9.3.5 (and from BFS strategy of RMC), we get that our algorithm is sound in proving Φ is satisfiable. □

9.5 Extensions

In this section, we discuss how to extend the RMC-based framework to support the following two types of *atomic string constraints*:

(i) A *length constraint* φ_i is a formula of Presburger arithmetic over the values of $|x|$ for $x \in \mathbb{X}$, where $|\cdot| : \mathbb{X} \to \omega$ 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 NFA representing a regular language.

9.5.1 Length Constraints

In order to extend our framework to solve word equations with length constraints, we encode them as regular languages. See Section 6.3 for all necessary details. Recall that a model $\sigma : \mathbb{X} \to \omega$ of a Presburger formula $\varphi_i(\mathbb{X})$ is encoded as a word w_σ over $\Sigma_{\mathbb{X}}$ expressing each assigned integer value in the binary form (note that there are multiple encodings). Regular language encoding both string constraints over $\mathbb{X}$ and length constraints $\varphi_i(\mathbb{X})$ contains all possible encodings of models of φ_i. The adjusted transformation relation then needs to reflect the effect of each transformation rule on the encoded length constraints. In the following paragraphs, we provide the details about our approach.

Consider a word w_σ encoding an assignment σ. The transformation $x \rightarrowtail yx$ for $x, y \in \mathbb{X}$ applied on w_σ produces a word $w_{\sigma'}$ encoding the assignment $\sigma' = \sigma \lhd \{x \mapsto \sigma(x) - \sigma(y)\}$ if $\sigma(x) \geq \sigma(y)$. 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 $\sigma(x) \geq 1$. Finally, the transformation $x \rightarrowtail \epsilon$ does not change the word w_σ, but imposes the restriction $\sigma(x) = 0$. Formally, the transformations are given as

$$
\begin{aligned}
T^{len}_{x \rightarrowtail yx} &= \{\, (w_\sigma, w_{\sigma'}) \mid \sigma(x) \geq \sigma(y) \wedge \sigma' = \sigma \lhd \{x \mapsto \sigma(x) - \sigma(y)\} \},\\
T^{len}_{x \rightarrowtail ax} &= \{\, (w_\sigma, w_{\sigma'}) \mid \sigma(x) \geq 1 \wedge \sigma' = \sigma \lhd \{x \mapsto \sigma(x) - 1\} \}, \text{ and}\\
T^{len}_{x \rightarrowtail \epsilon} &= \{\, (w_\sigma, w_\sigma) \mid \sigma(x) = 0 \,\}.
\end{aligned}
\tag{9.19}
$$

Lemma 9.5.1 shows that the transformations are regular. The proof is based on constructing MSO(Str) formulae with free variables ℓ, ℓ' implementing the transformations.

Lemma 9.5.1. *The relations $T^{len}_{x \rightarrowtail yx}$, $T^{len}_{x \rightarrowtail ax}$, and $T^{len}_{x \rightarrowtail \epsilon}$ are regular.*

$$\mathcal{I}^{len}_{\varphi_i} = \mathcal{I}^{eq}.\{\#_{len}\}.\mathcal{I}_{\varphi_i} \qquad \mathcal{D}^{len}_{\varphi_i} = \mathcal{D}^{eq}.\{\#_{len}\}.\Sigma^*_{\mathbb{X}}$$

$$\mathcal{T}^{len}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, y \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail yx}\right).\{\#_{len} \mapsto \#_{len}\}.T^{len}_{x \rightarrowtail yx} \cup \bigcup_{x \in \mathbb{X}, a \in \Sigma} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail ax}\right).\{\#_{len} \mapsto \#_{len}\}.T^{len}_{x \rightarrowtail ax}$$

$$\mathcal{T}^{len}_{x \rightarrowtail \epsilon} = \bigcup_{x \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \epsilon}\right).\{\#_{len} \mapsto \#_{len}\}.T^{len}_{x \rightarrowtail \epsilon}$$

Figure 9.10: RMC instantiation for a quadratic equation with a length constraint.

Proof. (Sketch) The transducer for $T^{len}_{x \rightarrowtail yx}$ can be straightforwardly constructed from the automaton $\mathcal{A}_\psi$ representing the Presburger formula $\psi \triangleq x' = x - y$ (see Section 6.3.3). The remaining relations $T^{len}_{x \rightarrowtail ax}$, and $T^{len}_{x \rightarrowtail \epsilon}$ can be constructed analogically. □

Let us now focus on how to adjust the initial and destination sets for an equation with a length constraint $\varphi_i(\mathbb{X})$, represented as a Presburger formula with free variables $\mathbb{X}$. The initial set is extended by all encoded models of φ_i. Formally, a part of the initial set related to the length constraint is given as $\mathcal{I}_{\varphi_i} = \mathcal{L}(\varphi_i)$ and a part of the destination set as $\mathcal{D}_{len} = \Sigma^*_{\mathbb{X}}$. Moreover, from Section 6.3.3 we have that $\mathcal{I}_{\varphi_i}$ is regular.

Satisfiability of a quadratic equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ with the length constraint φ_i can be then 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 3 with items given in Figure 9.10. Note the use of a fresh delimiter $\#_{len}$. Rationality of $\mathcal{T}^{len}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{len}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 9.1.1. The soundness of our algorithm is summarized by Lemma 9.5.2.

Lemma 9.5.2. *Given a quadratic word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ with the length constraint φ_i, Algorithm 3 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.*

Proof. (Sketch) We can generalize nodes of the Nielsen proof graph to pairs of the form $(\mathsf{t}'_\ell = \mathsf{t}'_r, f)$ where f is a mapping assigning lengths to variables from $\mathbb{X}$ (see, e.g., [191]). The transformation rules can be straightforwardly generalized to take into account also the lengths. The initial nodes are pairs $(\mathsf{t}_\ell = \mathsf{t}_r, f)$ 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 corresponds to $\mathcal{D}^{len}_{\varphi_i}$. You can also see that the transformation rules corresponds to the encoded relations $\mathcal{T}^{len}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{len}_{x \rightarrowtail \epsilon}$. Since the search in Algorithm 3 implements BFS strategy, we get that our (semi-)algorithm is sound in proving satisfiability. □

The satisfiability of a word equation *eq* with length constraints can be straightforwardly generalized to a system of equations Φ with length constraints. The languages/relations corresponding to *eq* are replaced by languages/relations corresponding to Φ. Note that Lemma 9.5.2 holds also for a system of equations.

9.5.2 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 NFA $\mathcal{A}$ (note that the negation of an atom $x \notin \mathcal{L}(\mathcal{A})$ can be converted to the positive atom $x \in \mathcal{L}(\mathcal{A}^{\complement})$. In particular, we assume that regular constraints are represented by a conjunction φ_r of ℓ atoms of the form

$$\varphi_r \triangleq \bigwedge_{i=1}^{\ell} x_i \in \mathcal{L}(\mathcal{A}_i), \tag{9.20}$$

where $\mathcal{A}_i$ is an NFA for each $1 \leq i \leq \ell$. Without loss of generality, we assume that the automata occurring in φ_r have pairwise disjoint sets of states and further we use $\mathcal{A}_r = (Q, \Sigma, \delta, I, F)$ to denote the disjoint union of all automata occurring in regular constraints.

We encode regular constraints as words over symbols of the form $\langle x, p, q\rangle$ where $x \in \mathbb{X}$ and $p, q \in Q$. We denote the set of all symbols as $\mathbb{X}_{\Sigma,\mathcal{A}_r}$. 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 $w \in \mathbb{X}^*_{\Sigma,\mathcal{A}_r}$ we use $w_{\preccurlyeq}$ to denote the string containing symbols sorted by $\preccurlyeq$ with no repetitions of symbols. A single atom $x \in \mathcal{L}(\mathcal{A})$ can be encoded as a set of words $\mathsf{encode}(x \in \mathcal{L}(\mathcal{A})) = \{\langle x, p, q\rangle \mid p \in I[\mathcal{A}], q \in F[\mathcal{A}]\}$. The set represents all possible accepting paths in $\mathcal{A}$. The initial set $\mathcal{I}_{\varphi_r}$ is then defined as

$$\mathcal{I}_{\varphi_r} = \{w_{\preccurlyeq} \mid w \in \mathsf{encode}(x_1 \in \mathcal{L}(\mathcal{A}_1)) \dots \mathsf{encode}(x_\ell \in \mathcal{L}(\mathcal{A}_\ell))\}. \tag{9.21}$$

Note that $\mathcal{I}_{\varphi_r}$ is finite for finite $\mathbb{X}$, therefore it is a regular language.

Let us now describe the effect of Nielsen transformation on the regular constraint part. Consider a word w encoding a set of symbols from $\mathbb{X}_{\Sigma,\mathcal{A}_r}$. Then, the transformation $x \rightarrowtail yx$ for $x, y \in \mathbb{X}$ applied on 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 $\langle y, p, r\rangle$ and $\langle x, r, q\rangle$ where $p \rightsquigarrow r$ and $r \rightsquigarrow q$ in $\mathcal{A}_r$. Similarly, the transformation $x \rightarrowtail ax$ for $x \in \mathbb{X}, a \in \Sigma$ applied on 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 $exp_{x,y}(\sigma) = \{\, \langle y, p, r\rangle.\langle x, r, q\rangle \mid r \in Q, p \rightsquigarrow r \rightsquigarrow q \text{ in } \mathcal{A}_r \,\}$ if $\sigma = \langle x, p, q\rangle$, and $exp_{x,y}(\sigma) = \{\sigma\}$ otherwise. Similarly, we define the expansion $exp_{x,a}(\sigma) = \{\, \langle x, r, q\rangle \mid r \in Q, p \xrightarrow{a} r \rightsquigarrow q \text{ in } \mathcal{A}_r \,\}$ if $\sigma = \langle x, p, q\rangle$, and $exp_{x,a}(\sigma) = \{\sigma\}$ otherwise. Then, the transformations $x \rightarrowtail yx$, $x \rightarrowtail ax$ and $x \rightarrowtail \epsilon$ can be described by the following relations

$$\begin{aligned}
T^{reg}_{x \rightarrowtail yx} &= \{\, (w, u_{\preccurlyeq}) \mid u \in exp_{x,y}(w[1]) \dots exp_{x,y}(w[|w|]) \,\},\\
T^{reg}_{x \rightarrowtail ax} &= \{\, (w, u_{\preccurlyeq}) \mid u \in exp_{x,a}(w[1]) \dots exp_{x,a}(w[|w|]) \,\}, \text{ and}\\
T^{reg}_{x \rightarrowtail \epsilon} &= \Big\{\, (w, w) \mid \forall 1 \leq i \leq |w| : \forall p, q \in Q : w[i] = \langle x, p, q\rangle \Rightarrow p = q \,\Big\}.
\end{aligned} \tag{9.22}$$

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.

Lemma 9.5.3. *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 $\mathbb{X}_{\Sigma,\mathcal{A}_r}$ to a total order $\mathbb{X}_{\Sigma,\mathcal{A}_r} \cup \{\square\}$ s.t. $\forall \sigma \in \mathbb{X}_{\Sigma,\mathcal{A}_r} : \sigma \preccurlyeq \square$. 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 analogically to $T^{reg}_{x \rightarrowtail yx}$.

$$\begin{aligned}\psi^{reg}_{x \rightarrowtail yx}(w, w') \triangleq \exists^{\mathbb{W}} u_1, u_2, u_3 (&\mathit{filter}_x(u_1, u_2, w) \\ &\wedge \mathit{expand}^y_x(u_1, u_3) \wedge \mathit{union}(u_2, u_3, w') \wedge \mathit{ordSet}(w'))\end{aligned} \tag{9.23}$$

$$\psi^{reg}_{x \rightarrowtail \epsilon}(w, w') \triangleq \forall^{\mathbb{P}} i \Big(w[i] = w'[i] \wedge \bigvee_{\substack{\xi \in (\mathbb{X} \setminus \{x\})_{\Sigma,\mathcal{A}_r} \cup \\ \{(x,q,q) \mid q \in Q\}}} w'[i] = \xi \Big) \tag{9.24}$$

where $\mathit{filter}_x(u, v, w)$ partitions symbols of w to u and v s.t. u contains symbols that are of the form $\langle x, -, - \rangle$ and v contains the remaining ones, $\mathit{expand}^y_x(u, v)$ replaces each symbol $\langle x, p, q \rangle$ in u with $\langle y, p, r \rangle$ and $\langle x, r, q \rangle$, and *union* is a set-like union. These predicates are defined as

$$\sigma \in w \triangleq \exists^{\mathbb{P}} i (w[i] = \sigma) \tag{9.25}$$

$$\mathit{set}(u) \triangleq \bigwedge_{\xi \in \mathbb{X}_{\Sigma,\mathcal{A}_r}} \forall^{\mathbb{P}} i, j (i \neq j \to (u[i] \neq \xi \vee u[j] \neq \xi)) \tag{9.26}$$

$$\mathit{ordSet}(u) \triangleq \mathit{set}(u) \wedge \forall^{\mathbb{P}} i, j (i < j \to u[i] \preccurlyeq u[j]) \tag{9.27}$$

$$\begin{aligned}\mathit{filter}_x(u, v, w) \triangleq \forall^{\mathbb{P}} i \Bigg(&\bigwedge_{\substack{q,r \in Q \\ \xi = \langle x,q,r \rangle}} (w[i] = \xi \to (u[i] = \xi \wedge v[i] = \square)) \\ &\wedge \bigwedge_{\substack{z \in \mathbb{X}_\Sigma, q,r \in Q \\ z \neq x, \xi = \langle z,q,r \rangle}} (w[i] = \xi \to (u[i] = \square \wedge v[i] = \xi)) \Bigg)\end{aligned} \tag{9.28}$$

$$\begin{aligned}\mathit{expand}^y_x(u, v) \triangleq &\bigwedge_{\substack{r,q \in Q, r \rightsquigarrow q \\ \xi' = \langle x,r,q \rangle}} \Big(\xi' \in v \to \bigvee_{\substack{p \in Q, p \rightsquigarrow r \\ \xi = \langle x,p,q \rangle \\ \xi'' = \langle y,p,r \rangle}} \xi \in u \wedge \xi'' \in v \Big) \\ &\wedge \bigwedge_{\substack{p,r \in Q, p \rightsquigarrow r \\ \xi'' = \langle y,p,r \rangle}} \Big(\xi'' \in v \to \bigvee_{\substack{q \in Q, r \rightsquigarrow q \\ \xi = \langle x,p,q \rangle \\ \xi' = \langle x,r,q \rangle}} \xi \in u \wedge \xi' \in v \Big) \\ &\wedge \bigwedge_{\substack{p,q \in Q \\ \xi = \langle x,p,q \rangle}} \Big(\xi \in u \to \bigvee_{\substack{r \in Q, p \rightsquigarrow r \rightsquigarrow q \\ \xi'' = \langle y,p,r \rangle \\ \xi' = \langle x,r,q \rangle}} \xi'' \in u \wedge \xi' \in v \Big)\end{aligned} \tag{9.29}$$

$$\mathit{union}(u, v, w) \triangleq \bigwedge_{\xi \in \mathbb{X}_{\Sigma,\mathcal{A}_r}} \xi \in w \leftrightarrow (\xi \in u \vee \xi \in v) \tag{9.30}$$

We further consider the relations $\tau_{+\mathsf{pad}} = \{(w, w') \mid w \in (\mathbb{X}_{\Sigma,\mathcal{A}_r} \cup \{\square\})^*, w' \in w.\{\square\}^*\}$ and $\tau_{-\mathsf{pad}} = \{(w, w') \mid w' \in (\mathbb{X}_{\Sigma,\mathcal{A}_r} \cup \{\square\})^*, w \in w'.\{\square\}^*\}$ appending and removing padding, respectively. These relations are rational. Then, observe that $\mathsf{pad}_\square(T^{reg}_{x \rightarrowtail yx}) = \tau_{+\mathsf{pad}} \circ \tau_{-\mathsf{pad}} \circ \mathcal{L}\left(\psi^{reg}_{x \rightarrowtail yx}\right) \circ \tau_{+\mathsf{pad}}$. From Propositions 9.1.1 and 9.1.2, we have that $\mathsf{pad}_\square(T^{reg}_{x \rightarrowtail yx})$ is rational (the same for $\mathsf{pad}_\square(T^{reg}_{x \rightarrowtail \epsilon})$). □

$$\mathcal{I}^{reg}_{\varphi_r} = \mathcal{I}^{eq}.\{\#_{reg}\}.\mathsf{pad}_{\square}(\mathcal{I}_{\varphi_r}) \qquad \mathcal{D}^{reg}_{\varphi_r} = \mathcal{D}^{eq}.\{\#_{reg}\}.\mathsf{pad}_{\square}(\mathcal{D}_{reg})$$

$$\mathcal{T}^{reg}_{x \rightarrowtail \alpha x} = \bigcup_{x \in \mathbb{X}, y \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail yx}\right).\{\#_{reg} \mapsto \#_{reg}\}.\mathsf{pad}_{\square}(T^{reg}_{x \rightarrowtail yx}) \cup \bigcup_{x \in \mathbb{X}, a \in \Sigma} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail ax}\right).\{\#_{reg} \mapsto \#_{reg}\}.\mathsf{pad}_{\square}(T^{reg}_{x \rightarrowtail ax})$$

$$\mathcal{T}^{reg}_{x \rightarrowtail \epsilon} = \bigcup_{x \in \mathbb{X}} \mathsf{encode}\left(\tau^{\leq 2}_{x \rightarrowtail \epsilon}\right).\{\#_{reg} \mapsto \#_{reg}\}.\mathsf{pad}_{\square}(T^{reg}_{x \rightarrowtail \epsilon})$$

Figure 9.11: RMC instantiation for a quadratic equation with a regular constraint.

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 $L^x = \{ w_{\preccurlyeq} \mid w = \langle x, q_1, r_1\rangle \cdots \langle x, q_n, r_n\rangle \in \mathbb{X}^*_{\Sigma,\mathcal{A}_r}, \bigcap_{i=1}^{n} \mathcal{L}_{\mathcal{A}_r}(q_i, r_i) \neq \emptyset \}$. Then, the destination set for a set of variables $\mathbb{X} = \{x_1, \ldots, x_k\}$ 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 $eq : \mathsf{t}_\ell = \mathsf{t}_r$ with a regular constraint φ_r can be expressed in the RMC framework $(\mathcal{I}^{reg}_{\varphi_r}, \mathcal{T}^{reg}_{x \rightarrowtail \alpha x} \cup \mathcal{T}^{reg}_{x \rightarrowtail \epsilon}, \mathcal{D}^{reg}_{\varphi_r})$ instantiating Algorithm 3 with items given in Figure 9.11. Note that we use a fresh delimiter $\#_{reg}$. The rationality of $\mathcal{T}^{reg}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}^{reg}_{x \rightarrowtail \epsilon}$ follows directly from Proposition 9.1.1. The soundness and completeness of our procedure is summarized by the following lemma.

Lemma 9.5.4. *Given a quadratic word equation $eq : \mathsf{t}_\ell = \mathsf{t}_r$ with a regular constraint φ_r, Algorithm 3 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 and complete.*

Proof. (Sketch) Similarly to proof of Lemma 9.5.2, we can generalize nodes of the Nielsen proof graph to pairs of the form $(\mathsf{t}'_\ell = \mathsf{t}'_r, S)$ where $S \subseteq \mathbb{X}_{\Sigma,\mathcal{A}_r}$. The transformation rules can be straightforwardly generalized to take into account also the regular constraints represented by a subset of $\mathbb{X}_{\Sigma,\mathcal{A}_r}$. Since $\mathbb{X}_{\Sigma,\mathcal{A}_r}$ is finite and *eq* is quadratic, the generalized proof graph is finite (see, e.g., [191]). The initial nodes of the generalized proof graph are exactly encoded strings from $\mathcal{I}^{reg}_{\varphi_r}$, the final nodes corresponds 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 BFS strategy, from the previous we get that our procedure is sound and complete 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 string equations with regular constraints our algorithm is still sound.

9.6 Implementation

We created a prototype Python tool called Retro, where we implemented the symbolic procedure for solving systems of word equations. Retro implements a modification of the RMC loop from Algorithm 3. In particular, instead of standard transducers defined in Section 9.1, it uses the so-called *finite-alphabet register transducers* (FRTs), which allow a more concise representation of a rational relation.

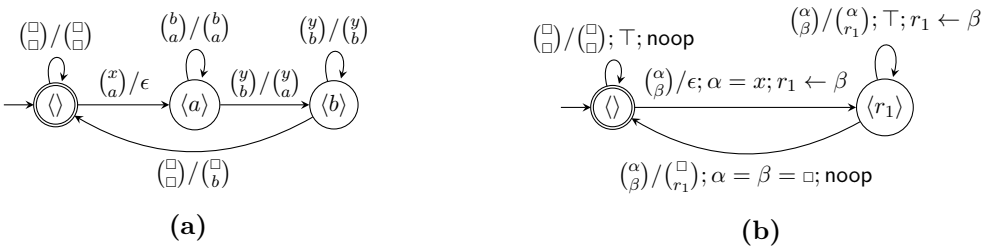


Figure 9.12: A partial (explicit) transducer (a) and FRT (b) implementing the encoded relation $\tau^{\leq 1}_{x \mapsto \epsilon}$. In the case of FRT, α, β are variable representing an input symbol, r_1 is a register, and the transitions are of the form *action;condition;register update*.

Informally, an FRT is a register automaton (in the sense of [162]) 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.

In particular, transducers (without registers) corresponding to the transformers $\mathcal{T}_{x \rightarrowtail \alpha x}$ and $\mathcal{T}_{x \rightarrowtail \epsilon}$ 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 an incomplete transducer in Figure 9.12a implementing a part of the encoded relation $\tau^{\leq 1}_{x \mapsto \epsilon}$. The full transducer would require branching for each $\binom{u}{v}$ and a lot of states to store concrete shifted symbols. On the other hand the partial FRT in Figure 9.12b stores the shifted symbols in the register r_1, the branching is replaced by a symbolic transition, and hence it requires less states and transitions (the full FRT would require another register to store the variable to replace).

As another feature, Retro uses deterministic finite automata to represent configurations in Algorithm 3. 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.

9.7 Experimental Evaluation

We compared the performance of our approach (implemented in Retro) with two current state-of-the-art SMT solvers that support the string theory: Z3 4.8.7 [218] and CVC4 1.7 [31].

The first set of benchmarks is $\texttt{Kepler}_{22}$, obtained from [184]. $\texttt{Kepler}_{22}$ contains 600 hand-crafted string constraints composed of quadratic word equations with length constraints. In Figure 9.13, we give a cactus plot of the results of the solvers on the $\texttt{Kepler}_{22}$ benchmark set with the timeout of 20 s. The total numbers of solved benchmarks within the timeout were: 119 for Z3, 266 for CVC4, and 443 for Retro (out of which 179 could not be solved by CVC4). On this benchmark set, Retro can solve significantly more benchmarks than both Z3 and CVC4.

The other set of benchmarks that we tried is PyEx-Hard. Here we want 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

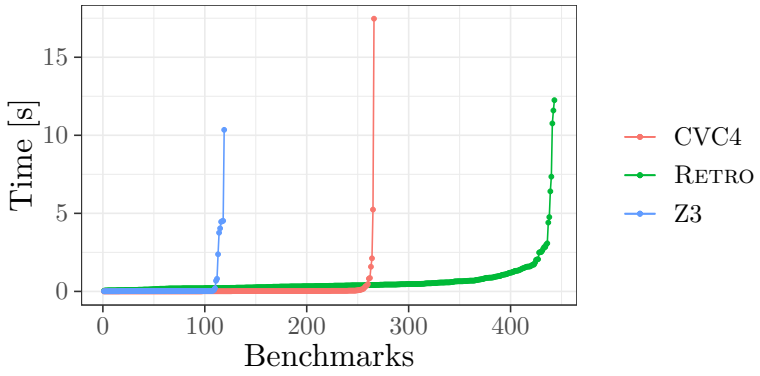


Figure 9.13: A cactus plot comparing Retro, CVC4, and Z3 on the $\texttt{Kepler}_{22}$ benchmark

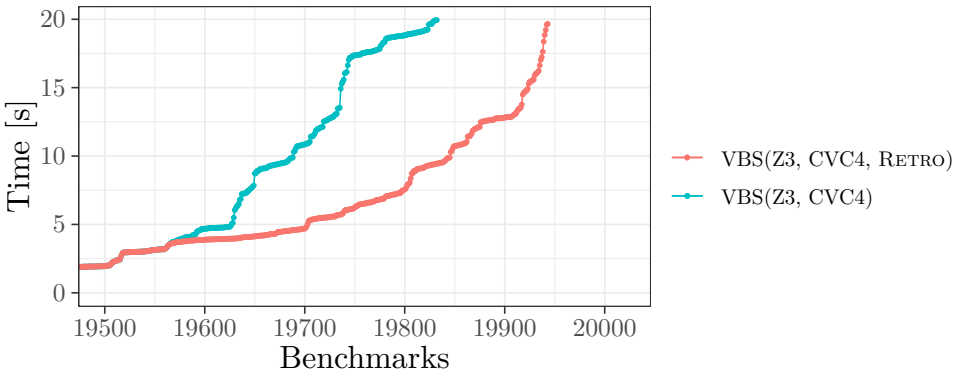


Figure 9.14: A cactus plot comparing the Virtual Best Solver with and without Retro on the PyEx-Hard benchmark. We show ∼500 most difficult benchmarks (from 20,020).

formulae (e.g., $xy = zb \wedge z = ax$) 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 [244] and extracted from it 967 difficult instances that neither CVC4 nor Z3 could solve in 10 seconds. 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 three solvers on PyEx-Hard with the timeout of 20 s. Out of these, Z3 could not solve 3,232, CVC4 could not solve 188, and Retro could not solve 3,099 instances.

Let us now closely look at the hard instances in the PyEx-Hard benchmark set, in particular, on the instances that either CVC4 or Z3 could not solve. These benchmarks cannot be handled by the (several layers of) fast heuristics implemented in CVC4 and Z3, which are sufficient to solve many benchmarks without the need to start applying the

case-split rule.[1] The set contains the 3,232 benchmarks that Z3 could not solve within 20 seconds. Out of these, CVC4 could not solve 188 benchmarks (CVC4 could solve every constraint that Z3 could solve), and Retro could not solve 568 benchmarks. When we compared the solvers on the examples that Z3 and CVC4 failed to solve, Retro could solve 2,664 examples (82.4 %) out of those where Z3 failed and 111 examples (59.04 %) of those where CVC4 failed. In Figure 9.14, we give a cactus plot of the *Virtual Best Solver* on the benchmarks with and without Retro. Given a set of solvers S, we use $VBS(S)$ to denote the solver that would be obtained by taking, for each benchmark, the solver that is the fastest on the given benchmark. The graph shows that our approach can significantly help solvers deal with hard equations.

Discussion. From the obtained results, we see that our approach works well in *hard cases*, where the fast heuristics implemented in state-of-the-art solvers are not sufficient to quickly discharge a formula, 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 and CVC4 written in C++ by a Python tool in those cases. We believe that 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.

9.8 Conclusion

In this chapter, we presented an encoding of Nielsen transformation into the RMC framework. We proposed an efficient encoding of a proof graph using finite automata with the transition rules expressed by finite transducers. The experimental evaluation, based on a prototype tool implementing register transducers with eager minimization of NFAs, shows a potential of the approach. A direction of a further research may include encoding of a complete procedure for solving string equations into the RMC framework (e.g., Makanin's algorithm [197]). The content of this chapter was published in the proceedings of APLAS'20 [76] and it was extended by encoding of regular a length constraints.

[1] 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.

Part III:
Büchi Automata Complementation

Chapter 10

Büchi Automata Complementation

Complementation of Büchi automata is an important and challenging problem since the time Büchi introduced his automaton model over infinite words in the context of a decision procedure for S1S [65] (see Section 6.4 for more details). Thenceforth, efficient complementation of BAs became a topic of great interest from both theoretical and practical angles. Büchi automata complementation is crucial for language inclusion checking of BAs, which can be used for model checking where both the system and the property of interest are represented by automata. Further, termination checking of programs as implemented within, e.g., the Ultimate Automizer tool, uses complementation to remove a set of traces whose termination has been established from the set of traces whose termination is yet to be established [142, 77]. Last but not least, complementation is a crucial operation for decision procedures of various logics, such as the aforementioned S1S or the temporal logics ETL and QPTL [255].

Büchi with his double-exponential complementation procedure launched a haunt for efficient complementation techniques. In 1988 Safra proposed a complementation approach based on intermediate deterministic Rabin automata with the upper bound $n^{\mathcal{O}(n)}$ [248]. Simultaneously to the efforts put into finding more efficient techniques, there appeared works aiming at establishing the lower bound of BA complementation. In contrast to the complementation of NFAs, where for an NFA having n states, an automaton accepting the complementary language has at most 2^n states, the situation is much more involved for Büchi automata. First, Michel [211] showed that there are BAs for which the lower bound of complementation is $n! = 2^{\Omega(n \log n)}$ (approximately $(0.36n)^n$). This result was later refined by Yan to $(0.76n)^n$ [294]. Since the construction of Safra asymptotically matches the lower bound, the problem seemed solved. On the other hand, the factor in the exponent plays a crucial role because a higher factor apparently affects the size of a complemented automaton. As a consequence, it may be limiting in real-world applications. Reducing this complexity gap became a subject of many works [182, 115, 281, 161, 294]. The efforts lead to the improved rank-based construction of Schewe [250] matching the lower bound of BA complementation modulo a $\mathcal{O}(n^2)$ polynomial factor.

In this chapter, we give a brief introduction to complementation techniques with focus on the rank-based complementation as we deal with its optimizations in the following chapters. The content of this chapter is partially based on [75, 139].

Chapter outline. This chapter serves as a brief introduction to BA complementation approaches, in particular to the rank-based complementation. In Section 10.1, we give definitions used in the following sections/chapters. In Section 10.2, we describe existing

complementation techniques and finally, in Section 10.3, we provide details about the rank-based complementation.

10.1 Preliminaries

In this section, we assume definitions related to functions, languages, and Büchi automata from Chapter 2. In this chapter, we, by $[n]$, denote the set $\{0, \ldots, n\}$ (do not confuse with the denotation of equivalence classes, which are specified by an equivalence relation, e.g., $[\cdot]_{\equiv}$). Further, let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a BA. For a pair of states p and q, and a word $w \in \Sigma^*$ we use $p \overset{w}{\rightsquigarrow} q$ to denote that q is reachable from p over the word w (we use the same notation as for NFAs); if a path from p to q over w contains an accepting state, we can write $p \overset{w}{\underset{F}{\rightsquigarrow}} q$.

Variants of ω-automata. In Chapter 2, we defined an accepting run of a BA over some infinite word to be a run with infinitely many accepting states. However, this accepting condition is not the only possible one. We can distinguish different automata models over infinite words (called ω-automata) based on various accepting conditions. We provide a couple of them. In particular, an ω-automaton $\mathcal{A} = (Q, \Sigma, \delta, I, Acc)$ is called

- a *Generalized Büchi automaton* if $Acc = \mathcal{F} \subseteq 2^Q$. A run ρ of $\mathcal{A}$ on a word $\alpha \in \Sigma^\omega$ is called accepting iff $\forall F \in \mathcal{F} : \inf(\rho) \cap F \neq \emptyset$.
- a *Muller automaton* if $Acc = \mathcal{F} \subseteq 2^Q$. A run ρ of $\mathcal{A}$ on a word $\alpha \in \Sigma^\omega$ is called accepting iff $\inf(\rho) \in \mathcal{F}$.
- a *Rabin automaton* if $Acc = \mathcal{F} \subseteq 2^Q \times 2^Q$. A run ρ of $\mathcal{A}$ on a word $\alpha \in \Sigma^\omega$ is called accepting iff $\exists (E, F) \in \mathcal{F} : \inf(\rho) \cap E = \emptyset \wedge \inf(\rho) \cap F \neq \emptyset$.
- a *Streett automaton* if $Acc = \mathcal{F} \subseteq 2^Q \times 2^Q$. A run ρ of $\mathcal{A}$ on a word $\alpha \in \Sigma^\omega$ is called accepting iff $\forall (E, F) \in \mathcal{F} : \inf(\rho) \cap E \neq \emptyset \vee \inf(\rho) \cap F = \emptyset$.

Note that for $Acc = F$ we get BAs defined in Chapter 2. Further, all the mentioned *nondeterministic* automata models (except the classical BAs) have the same expressive power [109].

10.2 Overview of the Complementation Techniques

As we already said in the introduction, complementation of Büchi automata is still an intensively studied problem. Currently, there are several branches of approaches based on slightly different way of reasoning. Although several of them are asymptotically optimal, their practical efficiency may differ [276]. In this section, we give a brief overview of the complementation techniques. For that, we assume a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$.

Ramsey-based complementation. A complementation technique based on the infinite Ramsey theorem with double-exponential upper bound was proposed already by Büchi [65]. The algorithm was later improved in [255]. We briefly describe the construction based on this work. The complemented automaton is constructed based on equivalences on Σ^ω using

generalized subset construction. For that, we first define the equivalence relation $\equiv$ on Σ^* as

$$x \equiv y \stackrel{\text{def}}{\equiv} \forall q, q' \in Q : (q \overset{x}{\rightsquigarrow} q' \Leftrightarrow q \overset{y}{\rightsquigarrow} q') \wedge (q \underset{F}{\overset{x}{\rightsquigarrow}} q' \Leftrightarrow q \underset{F}{\overset{y}{\rightsquigarrow}} q'). \tag{10.1}$$

Intuitively, if $x \equiv y$ then we can modify a word $\alpha \in \Sigma^\omega$ by replacing occurrences of x by y without affecting the acceptance/rejection of α in $\mathcal{A}$. Each equivalence class $[x]_\equiv$ can be represented by a finite automaton with the size bounded by $\mathcal{O}(4^{n^2})$. The index of $\equiv$ is bounded by 4^{n^2}. Further, for each $x, y \in \Sigma^*$ the language $L_{xy} = [x]_\equiv.[y]_\equiv^\omega$ is ω-regular, and moreover it holds that $L_{xy} \subseteq \mathcal{L}(\mathcal{A})$ or $L_{xy} \cap \mathcal{L}(\mathcal{A}) = \emptyset$. One can even restrict the languages to *proper* ones, i.e., languages L_{xy} s.t. $[x]_\equiv.[y]_\equiv \subseteq [x]_\equiv$ and $[y]_\equiv.[y]_\equiv \subseteq [y]_\equiv$. Using the infinite Ramsey theorem or results concerning monoids [227], it can be shown that each $\alpha \in \Sigma^\omega$ belongs to some proper L_{xy}. Putting it all together, we obtain

$$\overline{\mathcal{L}(\mathcal{A})} = \bigcup \{L_{xy} \mid x, y \in \Sigma^*, L_{xy} \text{ is proper}, L_{xy} \cap \mathcal{L}(\mathcal{A}) = \emptyset\}, \tag{10.2}$$

giving the way for a construction of the complemented automaton with the upper bound on the size $2^{\mathcal{O}(n^2)}$ [255]. This upper bound can be further improved by a preorder merging optimization yielding the size upper bound $2^{\mathcal{O}(n \log n)}$ [60].

Determinization-based complementation. Another approach for Büchi automata complementation is based on determinization. Bad news are that deterministic Büchi automata are strictly weaker that the nondeterministic ones, e.g., the language represented by the ω-regular expression $(a+b)^*b^\omega$ cannot be accepted by any deterministic Büchi automaton. However, nondeterministic BAs can be converted into an equivalent different ω-automaton model allowing determinization, e.g., Rabin, Muller, or Streett automata whose accepting power (even for deterministic variants) coincides with the family of ω-regular languages [109, 247]. The determinization-based complementation uses an equivalent intermediate deterministic automaton, which can be easily complemented, and then with some overhead translated back to a BA. In particular, the construction proposed by Safra [248] allows to construct a deterministic Rabin automaton $\mathcal{A}'$ equivalent to $\mathcal{A}$ having $2^{\mathcal{O}(n \log n)}$ states and $\mathcal{O}(n)$ accepting pairs. If we see $\mathcal{A}'$ by the optics of deterministic Streett automaton we have $\mathcal{L}(\mathcal{A}') = \overline{\mathcal{L}(A)}$. Moreover, every Streett automaton with m states and r accepting pairs can be converted into an equivalent BA with $\mathcal{O}(m \cdot 2^r)$ states [247]. The upper bound of the complementation is hence $2^{\mathcal{O}(n \log n)}$. The construction of Safra was later improved by Piterman [228] using intermediate deterministic parity automata with the n^{2n} upper bound.

Rank-based complementation. Rank-based complementation, introduced in [182] and later studied in [250, 115, 133, 163], tracks information about all runs over a word using generalized subset construction assigning numbers (ranks) to each state in a macrostate. The ranks allow to distinguish accepting and nonaccepting runs in the original BA. The improved construction of Schewe [250] matches the lower bound $(0.76)^n$ [294] of BA complementation modulo a polynomial factor of $\mathcal{O}(n^2)$. Heuristics and optimizations of the procedure were further studied in [133, 163]. Since the rank-based complementation is a fundamental stone for the following chapters, we provide a detailed explanation later in Section 10.3.

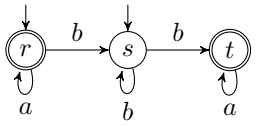


Figure 10.1: The BA $\mathcal{A}_{ex}$

Slice-based complementation. Slice-based complementation [281, 161] uses properties of reduced run trees. A run tree over a word α is a (possibly infinite) binary tree whose vertices are sets of states with I being the root. The left successor of a vertex v on a level i contains all final successor states over α_i of v, i.e., $\delta(v, \alpha_i) \cap F$. The right successor of v is then $\delta(v, \alpha_i) \setminus F$. The run trees may have unbounded width, therefore, the construction uses reduced run trees instead. A vertex v in a reduced tree contains only those states that do not occur in vertices on the left side of v on the same level.

The construction then simulates building of levels (slices) of a reduced run tree—states of the complemented automaton are (decorated) slices. In order to accept a word, it guesses at some point a slice s.t. all infinite paths in the reduced tree starting at this slice take right successors only. To verify the guess, vertices in a slice are decorated by flags to track the information about paths in a reduced run tree. For a BA having n states, the construction yields a BA having at most $(3n)^n$ states.

Other complementation techniques. In addition to techniques for complementation of general BAs, there are approaches for complementation of special variants of BAs. In particular, deterministic [183], semideterministic [47]—i.e., automata whose parts reachable from final states are deterministic, or unambiguous [188], i.e., for each word $\alpha \in \Sigma^\omega$ there is at most one accepting run.

From the other approaches for complementation of general BAs we mention an optimal algorithm by Allred and Utes-Nitsche [16] or a learning-based approach [187, 186] aiming at learning small BAs for a complementary language. Complementation of semideterministic automata is also used together with a combination of semideterminization (conversion of a BA into a semideterministic one) [85] for the complementation of general BAs [46].

10.3 Rank-based Complementation

Rank-based complementation is the main topic of Chapters 11 and 12. For this reason, in this section, we provide details about the construction. In the first part, we focus on the original rank-based construction. Then, we deal with the improved tight-ranking-based construction. In the last part, we recall the optimal construction of Schewe. We fix a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$.

10.3.1 Run DAGs

Before we move to the rank-based construction, we recall the terminology from [250] (which is a minor modification of the terminology from [182]), which we use heavily in the following sections and chapters. We fix the definition of the *run DAG* of $\mathcal{A}$ over a word α to be a DAG (directed acyclic graph) $\mathcal{G}_\alpha = (V, E)$ of vertices V and edges E where

- $V \subseteq Q \times \omega$ s.t. $(q, i) \in V$ iff there is a run ρ of $\mathcal{A}$ from I over α with $\rho_i = q$,

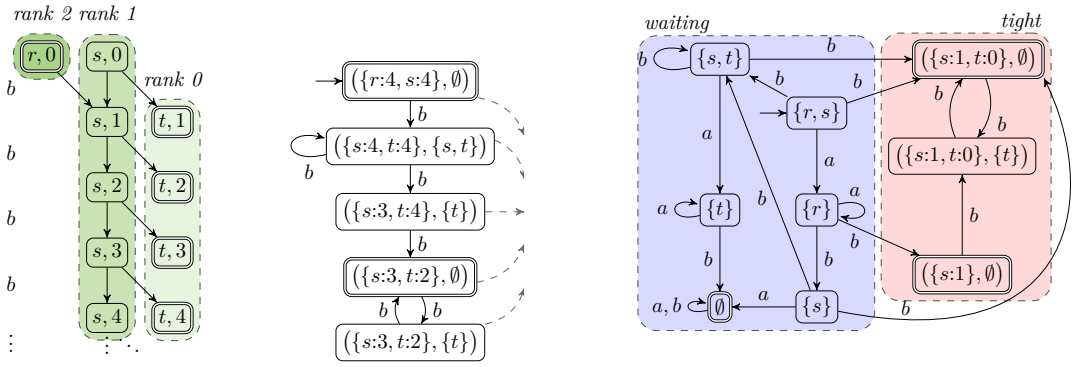


Figure 10.2: The run DAG of $\mathcal{A}_{ex}$ over b^ω (a). A part of $\mathrm{KV}(\mathcal{A}_{ex})$ (b). $\mathrm{FKV}(\mathcal{A}_{ex})$; the *waiting* and the *tight* parts are highlighted (c).

- $E \subseteq V \times V$ 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 $(p,i) \in V$. We call (p,i) *accepting* if p is an accepting state. $\mathcal{G}_\alpha$ is *rejecting* if it contains no path with infinitely many accepting vertices. 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 v cannot reach an accepting vertex.

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 is the number of states of $\mathcal{A}$.

- Set $\mathit{rank}_\alpha(v) := j$ 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 $\mathit{rank}_\alpha(v) := j+1$ 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 $j := j+2$.

For all vertices v that have not been assigned a rank yet, we assign $\mathit{rank}_\alpha(v) := \omega$. See Figure 10.1 for an example BA $\mathcal{A}_{ex}$ and Figure 10.2a for the run DAG of $\mathcal{A}_{ex}$ over b^ω. The properties of run DAGs are summarized by the following lemma.

Lemma 10.3.1 ([182])**.** *If $\alpha \notin \mathcal{L}(\mathcal{A})$, then $0 \leq \mathit{rank}_\alpha(v) \leq 2n$ for all $v \in \mathcal{G}_\alpha$. Moreover, if $\alpha \in \mathcal{L}(\mathcal{A})$, then there is a vertex $(p,0) \in \mathcal{G}_\alpha$ s.t. $\mathit{rank}_\alpha(p,0) = \omega$.*

10.3.2 Basic Rank-Based Complementation

The intuition behind rank-based complementation algorithms is that states in the complemented automaton $\mathcal{C}$ *track* all runs of the original automaton $\mathcal{A}$ on the given word and the possible *ranks* of each of the runs. Loosely speaking, an accepting run of a complement automaton $\mathcal{C}$ on a word $\alpha \notin \mathcal{L}(\mathcal{A})$ represents the run DAG of $\mathcal{A}$ over α (in the complement, each state in a *macrostate* is assigned a *rank*)[1].

[1]This is not entirely true since there may be more accepting runs of $\mathcal{C}$ over α, with ranks assigned to states of $\mathcal{A}$ that are higher than the ranks in the run DAG.

Rank-based complementation procedures work with the notion of level rankings of states of $\mathcal{A}$, originally proposed in [182, 115]. For $n = |Q|$, a *(level) ranking* is a function $f\colon Q \to [2n]$ such that $\{f(q_f) \mid q_f \in F\} \subseteq \{0, 2, \ldots, 2n\}$, i.e., f assigns even ranks to accepting states of $\mathcal{A}$. We use $\mathcal{R}$ to denote the set of all rankings and $\mathit{odd}(f)$ to denote the set of states given an odd ranking by f, i.e. $\mathit{odd}(f) = \{q \in Q \mid f(q) \text{ is odd}\}$. 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 $f < f'$ iff $f \leq f'$ and there is a state $p \in Q$ with $f(p) < f'(p)$.

The basic rank-based procedure, called KV, constructs the BA $\mathrm{KV}(\mathcal{A}) = (Q', \Sigma, \delta', I', F')$ whose components are defined as follows [182]:

- $Q' = 2^Q \times 2^Q \times \mathcal{R}$ is a set of *macrostates* denoted as (S, O, f),
- $I' = \{I\} \times \{\emptyset\} \times \mathcal{R}$,
- $(S', O', f') \in \delta'((S, O, f), a)$ iff
  - $S' = \delta(S, a)$,
  - for every $q \in S$ and $q' \in \delta(q, a)$ it holds that $f'(q') \leq f(q)$, and
  - $O' = \begin{cases} \delta(S, a) \setminus \mathit{odd}(f') \text{ if } O = \emptyset, \\ \delta(O, a) \setminus \mathit{odd}(f') \text{ otherwise, and} \end{cases}$
- $F' = 2^Q \times \{\emptyset\} \times \mathcal{R}$.

The macrostates (S, O, f) of $\mathrm{KV}(\mathcal{A})$ are composed of three components. The S component tracks all runs of $\mathcal{A}$ over the input word in the same way as determinization of an NFA. The O component, on the other hand, tracks all runs whose rank has been even since the last cut-point (a point where $O = \emptyset$). The last component, f, assigns every state in S a rank. Note that the f component is responsible for the nondeterminism of the complement (and also for the content of the O component). A run of $\mathrm{KV}(\mathcal{A})$ is accepting if it manages to empty the O component of states occurring on the run infinitely often. In the worst case, KV constructs a BA with approximately $(6n)^n$ states [182].

For a better readability, in the examples we often merge S and f components and use, e.g., $(\{r{:}4, s{:}4\}, \emptyset)$ to denote the macrostate $(\{r, s\}, \emptyset, \{r \mapsto 4, s \mapsto 4\})$ (we also omit ranks of states not in S). See Figure 10.2b for a part of $\mathrm{KV}(\mathcal{A}_{ex})$ that starts at $(\{r{:}4, s{:}4\}, \emptyset)$ and keeps ranks as high as possible (the whole automaton is prohibitively large to be shown here—the implementation of KV in GOAL [277] outputs a BA with 98 states). Note that in order to accept the word b^ω, the accepting run needs to nondeterministically decrease the rank of the successor of s (the transition $(\{s{:}4, t{:}4\}, \{s, t\}) \xrightarrow{b} (\{s{:}3, t{:}4\}, \{t\})$).

10.3.3 Complementation with Tight Rankings

Friedgut, Kupferman, and Vardi observed in [115] that the KV construction generates macrostates with many rankings that are not strictly necessary in the loop part of the lasso for an accepting run on a word. Their optimization, denoted as FKV, is based on composing the complement automaton from two parts: the first part (called by us the *waiting* part) just tracks all runs of $\mathcal{A}$ over the input word (in a similar manner as in a determinized NFA) and the second part (the *tight* part) in addition tracks the rank of each run in a similar manner as the KV construction, with the difference that the rankings are *tight*.

For a set of states $S \subseteq Q$, we call f to be *S-tight* if (i) it has an odd rank r, (ii) $\{f(s) \mid s \in S\} \supseteq \{1, 3, \ldots, r\}$, and (iii) $\{f(q) \mid q \notin S\} = \{0\}$. A ranking is *tight* if it is Q-tight; we use $\mathcal{T}$ to denote the set of all tight rankings. The FKV procedure constructs the BA $\text{FKV}(\mathcal{A}) = (Q', \Sigma, \delta', I', F')$ whose components are defined as follows:

- $Q' = Q_1 \cup Q_2$ where
  - $Q_1 = 2^Q$ and
  - $Q_2 = \{(S, O, f) \in 2^Q \times 2^Q \times \mathcal{T} \mid f \text{ is } S\text{-tight}, O \subseteq S\}$,
- $I' = \{I\}$,
- $\delta' = \tau_1 \cup \tau_2 \cup \tau_3$ where
  - $\tau_1 \colon Q_1 \times \Sigma \to 2^{Q_1}$ such that $\tau_1(S, a) = \{\delta(S, a)\}$,
  - $\tau_2 \colon Q_1 \times \Sigma \to 2^{Q_2}$ such that $\tau_2(S, a) = \{(S', \emptyset, f) \in Q_2 \mid S' = \delta(S, a), f \text{ is } S'\text{-tight}\}$, and
  - $\tau_3 \colon Q_2 \times \Sigma \to 2^{Q_2}$ such that $(S', O', f') \in \tau_3((S, O, f), a)$ iff
    - $S' = \delta(S, a)$,
    - for every $q \in S$ and $q' \in \delta(q, a)$ it holds that $f'(q') \leq f(q)$,
    - $\mathit{rank}(f) = \mathit{rank}(f')$, and
    - $O' = \begin{cases} \delta(S, a) \setminus \mathit{odd}(f') \text{ if } O = \emptyset, \\ \delta(O, a) \setminus \mathit{odd}(f') \text{ otherwise, and} \end{cases}$
- $F' = \{\emptyset\} \cup ((2^Q \times \{\emptyset\} \times \mathcal{T}) \cap Q_2)$.

We call the part of $\text{FKV}(\mathcal{A})$ with the states in Q_1 the *waiting* part and the part with the states in Q_2 the *tight* part (an accepting run in $\text{FKV}(\mathcal{A})$ simulates the run DAG of $\mathcal{A}$ over a word w by *waiting* in Q_1 until it can generate *tight rankings* only; then it moves to Q_2). See Figure 10.2c for $\text{FKV}(\mathcal{A}_{ex})$. Note that $\text{FKV}(\mathcal{A}_{ex})$ is significantly smaller than $\text{KV}(\mathcal{A}_{ex})$ (which had 98 states). In the worst case, FKV constructs a BA with $\mathcal{O}((0.96n)^n)$ states [115].

10.3.4 An Optimal Algorithm

An optimal complementation algorithm whose space complexity matches the theoretical lower bound $\mathcal{O}((0.76n)^n)$ was given by Schewe in [250]. We denote this algorithm as Schewe. The difference from FKV is that in Schewe, macrostates contain one additional component, i.e., a macrostate has the form (S, O, f, i), where the last component $i \in \{0, 2, \ldots, 2n-2\}$, for $n = |Q|$, denotes the rank of states that are in O. Then, at a cut-point (when O is being reset), O is not filled with all states having an even rank (as in FKV), but only those whose rank is i (at every cut-point, i changes to $i+2$ modulo the rank of f).

The procedure of [250], denoted as Schewe, constructs the BA $\text{Schewe}(\mathcal{A}) = (Q', \Sigma, \delta', I', F')$ 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
  - $\delta_1 : Q_1 \times \Sigma \to 2^{Q_1}$ such that $\delta_1(S, a) = \{\delta(S, a)\}$,
  - $\delta_2 : Q_1 \times \Sigma \to 2^{Q_2}$ such that $\delta_2(S, a) = \{(S', \emptyset, f, 0) \mid S' = \delta(S, a), f \text{ is } S'\text{-tight}\}$, and
  - $\delta_3 : Q_2 \times \Sigma \to 2^{Q_2}$ such that $(S', O', f', i') \in \delta_3((S, O, f, i), a)$ iff
    - $S' = \delta(S, a)$,
    - for every $q \in S$ and $q' \in \delta(q, a)$ it holds that $f'(q') \leq f(q)$,
    - $\mathit{rank}(f) = \mathit{rank}(f')$,
    - and
      - $i' = (i + 2) \mod (\mathit{rank}(f') + 1)$ and $O' = f'^{-1}(i')$ if $O = \emptyset$ or
      - $i' = i$ and $O' = \delta(O, a) \cap f'^{-1}(i)$ if $O \neq \emptyset$, and

- $F' = \{\emptyset\} \cup ((2^Q \times \{\emptyset\} \times \mathcal{T} \times \omega) \cap Q_2)$.

Schewe($\mathcal{A}_{ex}$) would look almost the same as FKV($\mathcal{A}_{ex}$) (see Figure 10.2c) with the difference that the macrostates in the tight part would have an additional $i = 0$ component ($\mathcal{A}_{ex}$ is too small to see any real effect). The correctness of the construction is then given by the following proposition.

Proposition 10.3.1. ([250]) *Let $\mathcal{B} = \textsc{Schewe}(\mathcal{A})$. Then $\mathcal{L}(\mathcal{B}) = \overline{\mathcal{L}(\mathcal{A})}$.*

In the following chapters, we propose optimizations aiming at a reduction of the number of states or transitions in Schewe's construction.

Chapter 11

Simulations in Rank-Based Büchi Automata Complementation

As we already mentioned in Chapter 10, Büchi automata complementation is a fundamental problem in program analysis and formal verification. We also gave an overview of the existing techniques for BA complementation, such as Ramsey-based, determinization-based, or rank-based. Moreover, we focused our attention on the optimal (modulo a factor of $\mathcal{O}(n^2)$) rank-based construction of Schewe [250]. Although the algorithm of Schewe is worst-case optimal, it often generates unnecessarily large complements.

The standard approach to alleviate this problem is to decrease the size of the input BA before the complementation starts. Since minimization of (nondeterministic) BAs is a **PSPACE**-complete problem, more lightweight reduction methods are necessary. The most prevalent approaches are those based on various notions of *simulation-based reduction*, such as reductions based on *direct simulation* [68, 259], the richer *delayed simulation* [107], or their *multi-pebble* variants [108]. These approaches first compute a simulation relation over the input BA—which can be done with the time complexity $\mathcal{O}(mn)$ [144, 153, 238, 239, 88] and $\mathcal{O}(mn^3)$ [107] for direct and delayed simulation, respectively, with the number of states n and transitions m—and then construct a *quotient* BA by merging simulation-equivalent states, while preserving the language of the input BA. The other approach is a reduction based on *fair simulation* [132]. The fair simulation cannot, however, be used for quotienting, but still it can be used for merging certain states and removing transitions. The reduced BA is used as the input of the complementation, which often significantly reduces the size of the result. In this chapter, we propose a way of how to exploit the direct and delayed simulations in BA complementation even further to obtain smaller complements and shorter running times.

Overview of the proposed approach. In this chapter, we focus, in particular, on a simulation-based optimization in the rank-based complementation procedure, denoted as Schewe (see Section 10.3.4). Assume that for an input BA $\mathcal{A}$ the output of the rank-based complementation procedure is the BA $\mathcal{B}$. The main contribution of this chapter is to use simulation relations to remove some useless macrostates during the construction of $\mathcal{B}$. In particular, if a state p is simulated by q in $\mathcal{A}$, this puts a restriction on the relation between the ranks of runs from p and from q. As a consequence, macrostates that assign ranks violating this restriction can be purged from $\mathcal{B}$.

The proposed optimization is orthogonal to simulation-based size reduction mentioned above. Since the quotienting methods are based on taking only the symmetric fragment of the simulation, i.e., they merge states that simulate *each other*, after the quotienting, there might still be many pairs where the simulation holds in only one way and can therefore be exploited by our technique. Since the considered notions of simulation-based quotienting preserve the respective simulations, our techniques can be used to optimize the complementation *at no additional cost*. Our experimental evaluation of the optimization showed that, in many cases, they indeed significantly reduce the size of the complemented BA.

Related work. Techniques for complementation of BAs are discussed in Chapter 10. In addition to that, we mention works aiming at optimizations in rank-based complementation. In particular, the work [133] contains optimizations of a rank-based construction from [182] that uses intermediate alternating Büchi automata (the construction itself is not optimal). Further, the work [163] proposes an optimization and a modification of Schewe. Although it can sometimes provide a smaller automaton, the construction is, however, not compatible with the optimization presented in this chapter.

Because of the high computational complexity of complementing a BA, and, consequently, also checking BA inclusion and universality (which use complementation as their component), there has been some effort to develop heuristics that help to reduce the number of explored states in practical cases. The most prominent ones are heuristics that leverage various notions of simulation relations, which often provide a good compromise between the overhead they impose and the achieved state space reduction. Direct [68, 259], delayed [107], fair [107], their variants for alternating Büchi automata [116], and multi-pebble simulations [108] are the best-studied relations of this kind. Some of the relations can be used for quotienting, but also for pruning transitions entering simulation-smaller states (which may cause some parts of the BA to become inaccessible). A series of results in this direction were recently developed by Clemente and Mayr [80, 206, 207].

Not only can the relations be used for reducing the size of the input BA, they can also be used for under-approximating inclusion of languages of states. For instance, during a BA inclusion test $\mathcal{L}(\mathcal{A}_S) \subseteq^? \mathcal{L}(\mathcal{A}_B)$, if every initial state of $\mathcal{A}_S$ is simulated by an initial state of $\mathcal{A}_B$, the inclusion holds and no complementation needs to be performed. But simulations can also be used to reduce the explored state space within, e.g., the inclusion check itself, for instance in the context of Ramsey-based algorithms [12, 13]. The way simulations are applied in the Ramsey-based approach is fundamentally different from the current work, which is based on rank-based construction. Taking universality checking as an example, the algorithm checks if the language of the complement automaton is empty. They run the complementation algorithm and the emptiness check together, on the fly, and during the construction check if a macrostate with a larger language has been produced before; if yes, then they can stop the search from the language-smaller macrostate. Note that, in contrast to our approach, their algorithm does not produce the complement automaton.

Chapter outline. This chapter is structure as follows. Section 11.1 deals with preliminary definitions and results related to simulation relations for BAs. In Section 11.2, we propose our optimization based on removing states with incompatible rankings. Section 11.3 discusses the use of our optimization after simulation-based reductions. Finally, Section 11.4 presents experimental evaluation and Section 11.5 concludes the chapter.

11.1 Simulations

In this chapter, we assume definitions from Chapters 2 and 10. We extend them for the need of this chapter. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a BA. For a pair of states p and q in $\mathcal{A}$, we use $p \subseteq_{\mathcal{L}} q$ to denote $\mathcal{L}_{\mathcal{A}}(p) \subseteq \mathcal{L}_{\mathcal{A}}(q)$. A *trace* over a word $\alpha \in \Sigma^\omega$ is an infinite sequence $\pi = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \cdots$ such that $\rho = q_0 q_1 \ldots$ is a run of $\mathcal{A}$ over α from q_0. We say π is *fair* if it contains infinitely many accepting states. For the sake of proofs, we assume $\mathcal{A}$ to be complete. In this chapter, we fix a complete BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$.

Simulations. In this paragraph, we introduce simulation relations between states of a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ using the game semantics in a similar manner as in the extensive study of Clemente and Mayr [206]. In particular, in a *simulation game* between two players (called Spoiler and Duplicator) in $\mathcal{A}$ from a pair of states (p_0, r_0), for any (infinite) trace over a word α that Spoiler takes starting from p_0, Duplicator tries to mimic the trace starting from r_0. On the other hand, Spoiler tries to find a trace that Duplicator cannot mimic. The game starts in the configuration (p_0, r_0) and every i-th round proceeds by, first, Spoiler choosing a transition $p_i \xrightarrow{\alpha_i} p_{i+1}$ and, second, Duplicator mimicking Spoiler by choosing a matching transition $r_i \xrightarrow{\alpha_i} r_{i+1}$ over the same symbol α_i. The next game configuration is (p_{i+1}, r_{i+1}). Suppose that $\pi_p = p_0 \xrightarrow{\alpha_0} p_1 \xrightarrow{\alpha_1} \cdots$ and $\pi_r = r_0 \xrightarrow{\alpha_0} r_1 \xrightarrow{\alpha_1} \cdots$ are the two (infinite) traces constructed during the game. Duplicator *wins* the simulation game if $\mathcal{C}^x(\pi_p, \pi_r)$ holds, where $\mathcal{C}^x(\pi_p, \pi_r)$ is a condition that depends on the particular simulation. In this chapter and in Chapter 12, we consider especially the following simulation relations:

- **direct simulation** [96]: $\mathcal{C}^{di}(\pi_p, \pi_r) \stackrel{\text{def}}{\equiv} \forall i : p_i \in F \Rightarrow r_i \in F$,
- **delayed simulation** [107]: $\mathcal{C}^{de}(\pi_p, \pi_r) \stackrel{\text{def}}{\equiv} \forall i : p_i \in F \Rightarrow \exists k \geq i : r_k \in F$, and
- **fair simulation** [145]: $\mathcal{C}^{f}(\pi_p, \pi_r) \stackrel{\text{def}}{\equiv}$ if π_p is fair, then π_r is fair.

A maximal x-simulation relation $\preceq_x \subseteq Q \times Q$, for $x \in \{di, de, f\}$, is defined such that $p \preceq_x r$ iff Duplicator has a winning strategy in the simulation game with the winning condition $\mathcal{C}^x$ starting from (p, r). Formally, we define a strategy to be a (partial) mapping $\sigma : Q \times (Q \times \Sigma \times Q) \to Q$ such that $\sigma(r, p \xrightarrow{a} p') \in \delta(r, a)$ if $\delta(r, a) \neq \emptyset$, i.e., if Duplicator is in state r and Spoiler selects a transition $p \xrightarrow{a} p'$, the strategy picks a state r' such that $r \xrightarrow{a} r' \in \delta$. Note that Duplicator cannot look ahead at Spoiler's future moves. We use σ_x to denote any winning strategy of Duplicator in the $\mathcal{C}^x$ simulation game. Let σ_x and σ'_x be a pair of winning strategies in the $\mathcal{C}^x$ simulation game. Strategies are also lifted to traces as follows: let π_p be as above, then $\sigma(r_0, \pi_p) = r_0 \xrightarrow{\alpha_0} r_1 \xrightarrow{\alpha_1} \cdots$ if for all $i \leq 0$ it holds that $\sigma(r_i, p_i \xrightarrow{\alpha_i} p_{i+1}) = r_{i+1}$, otherwise it is undefined. The considered simulation relations form the following hierarchy: $\preceq_{di} \subseteq \preceq_{de} \subseteq \preceq_f \subseteq \subseteq_{\mathcal{L}}$. Note that every maximal simulation relation is a preorder, i.e., reflexive and transitive. Also note that direct simulation for BAs corresponds to forward simulation for NFAs described in Section 3.3.2.

11.2 Purging Macrostates with Incompatible Rankings

Our optimization is based on removing from Schewe($\mathcal{A}$) macrostates $(S, O, f, i) \in Q_2$ whose level ranking f assigns some states of S an unnecessarily high rank. Intuitively, when S contains a state p and a state q such that p is (directly) simulated by q, i.e. $p \preceq_{di} q$,

then $f(p)$ needs to be at most $f(q)$. This is because in any word α and its run DAG $\mathcal{G}_\alpha$ in $\mathcal{A}$, if p and q are at the same level i of $\mathcal{G}_\alpha$, then the ranks of their vertices v_p and v_q at the given level are either both ω (when $\alpha \in \mathcal{L}(\mathcal{A})$), or such that $rank_\alpha(v_p) \leq rank_\alpha(v_q)$ otherwise. This is because, intuitively, the DAG rooted in v_p in $\mathcal{G}_\alpha$ is isomorphic to a subgraph of the DAG rooted in v_q.

Formally, consider the following predicate on macrostates of SCHEWE($\mathcal{A}$):

$$\mathcal{P}_{di}(S, O, f, i) \stackrel{\text{def}}{\equiv} \exists p, q \in S : p \preceq_{di} q \wedge f(p) > f(q). \tag{11.1}$$

We modify SCHEWE to purge macrostates that satisfy $\mathcal{P}_{di}$. That is, we create a new procedure PRG$_{di}$ obtained from SCHEWE by modifying the definition of SCHEWE($\mathcal{A}$) such that all occurrences of Q_2 are substituted by Q_2^{di} and

$$Q_2^{di} = Q_2 \setminus \{(S, O, f, i) \in Q_2 \mid \mathcal{P}_{di}(S, O, f, i)\}. \tag{11.2}$$

The following lemma, proved in Section 11.2.1 states the correctness of this construction.

Lemma 11.2.1. $\mathcal{L}(PRG_{di}(\mathcal{A})) = \mathcal{L}(SCHEWE(\mathcal{A}))$

The following natural question arises: Is it possible to extend the purging technique from direct simulation to other notions of simulation? For *fair* simulation, this cannot be done. The reason is that, for a pair of states p and q s.t. $p \preceq_f q$, it can happen that for a word $\beta \in \Sigma^\omega$, there can be a trace from p over β that finitely many times touches an accepting state (i.e., a vertex of p in the corresponding run DAG can have any rank between 0 and $2n$), while all traces from q over β can completely avoid touching any accepting state. From the point of view of fair simulation, these are both unfair traces, and, therefore, disregarded.

On the other hand, *delayed* simulation—which is often much richer than direct simulation—can be used, with a small change. Intuitively, the delayed simulation can be used because $p \preceq_{de} q$ guarantees that on every level of trees in $\mathcal{G}_\alpha$ rooted in v_p and in v_q, respectively, the rank of the vertex v_p is at most by one larger than the rank of vertex v_q (or by any number smaller). Formally, let $\mathcal{P}_{de}$ be the following predicate on macrostates of SCHEWE($\mathcal{A}$):

$$\mathcal{P}_{de}(S, O, f, i) \stackrel{\text{def}}{\equiv} \exists p, q \in S : p \preceq_{de} q \wedge f(p) > \lceil\!\lceil f(q) \rceil\!\rceil, \tag{11.3}$$

where $\lceil\!\lceil x \rceil\!\rceil$ for $x \in \omega$ denotes the smallest even number greater or equal to x and $\lceil\!\lceil \omega \rceil\!\rceil = \omega$. Similarly as above, we create a new procedure, called PRG$_{de}$, which is obtained from SCHEWE by modifying the definition of SCHEWE($\mathcal{A}$) such that all occurrences of Q_2 are substituted by Q_2^{de} and

$$Q_2^{de} = Q_2 \setminus \{(S, O, f, i) \in Q_2 \mid \mathcal{P}_{de}(S, O, f, i)\}. \tag{11.4}$$

As an example, consider the BA in Figure 11.1a. PRG$_{de}$ optimization removes the macrostate $(\{p{:}1, q{:}2, r{:}3\}, \emptyset, 0)$ because $r \preceq_{de} p$ and $f(r) > \lceil\!\lceil f(p) \rceil\!\rceil$. The following lemma, proved in Section 11.2.1 states the correctness of this construction.

Lemma 11.2.2. $\mathcal{L}(PRG_{de}(\mathcal{A})) = \mathcal{L}(SCHEWE(\mathcal{A}))$

The use of $\lceil\!\lceil f(q) \rceil\!\rceil$ in $\mathcal{P}_{de}$ results in the fact that the two purging techniques are incomparable. For instance, consider a macrostate $(\{p{:}2, q{:}1\}, \emptyset, 0)$ such that $p \preceq_{di} q$ and $p \preceq_{de} q$. Then the macrostate will be purged in PRG$_{di}$, but not in PRG$_{de}$.

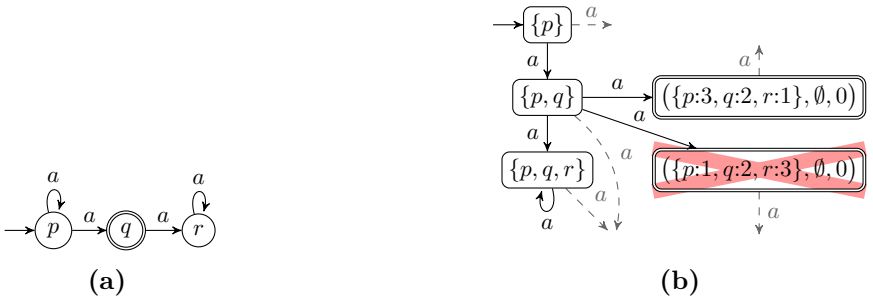


Figure 11.1: Illustration of PRG_{de} optimization (b) applied on a BA given in (a).

The two techniques can, however, be easily combined into a third procedure PRG_{di+de}, when Q_2 is substituted in SCHEWE with Q_2^{di+de} defined as

$$Q_2^{di+de} = Q_2 \setminus \{(S, O, f, i) \in Q_2 \mid \mathcal{P}_{di}(S, O, f, i) \vee \mathcal{P}_{de}(S, O, f, i)\}. \tag{11.5}$$

As in the previous cases, the following lemma, proved in Section 11.2.1 states the correctness of this construction.

Lemma 11.2.3. $\mathcal{L}\left(\text{PRG}_{di+de}(\mathcal{A})\right) = \mathcal{L}\left(\text{SCHEWE}(\mathcal{A})\right)$

11.2.1 Proofs of Lemmas 11.2.1, 11.2.2, and 11.2.3

We first give a lemma that an x-strategy σ_x preserves an x-simulation $\preceq_x$.

Lemma 11.2.4. *Let $\preceq_x$ be an x-simulation (for $x \in \{di, de, f\}$). Then, the following holds: $\forall p, q \in Q : p \preceq_x q \wedge p \xrightarrow{a} p' \in \delta \Rightarrow \exists q' \in Q : q \xrightarrow{a} q' \in \delta \wedge p' \preceq_x q'$.*

Proof. Let $p, q \in Q$ such that $p \preceq_x q$ and $p \xrightarrow{a} p' \in \delta$, and let π_p be a trace starting from p with the first transition $p \xrightarrow{a} p'$. From the definition of x-simulation, there is a winning Duplicator strategy σ_x; let $\pi_q = \sigma_x(q', \pi_p)$ and let $q \xrightarrow{a} q'$ be the first transition of π_q. Let $\pi_{p'}$ and $\pi_{r'}$ be traces obtained from π_p and π_r by removing their first transitions. It is easy to see that if $\mathcal{C}^x(\pi_p, \pi_r)$ then also $\mathcal{C}^x(\pi_{p'}, \pi_{r'})$ for any $x \in \{di, de, f\}$. It follows that σ_x is also a winning Duplicator strategy from (p', r'). □

Next, we focus on delayed simulation and the proof of Lemma 11.2.2. In the next lemma, we show that if there is a pair of vertices on some level of the run DAG where one vertex delay-simulates the other one, there exists a relation between their rankings. This will be used to purge some useless rankings from the complemented BA.

Lemma 11.2.5. *Let $p, q \in Q$ such that $p \preceq_{de} q$ and $\mathcal{G}_\alpha = (V, E)$ be the run DAG of $\mathcal{A}$ over α. For all $i \geq 0$, it holds that $(p, i) \in V \wedge (q, i) \in V \Rightarrow \mathit{rank}_\alpha(p, i) \leq \lVert \mathit{rank}_\alpha(q, i) \rVert$.*

Proof. Consider some $(p, i) \in V$ and $(q, i) \in V$. First, suppose that $\mathit{rank}_\alpha(q, i) = \omega$. Since the rank can be at most ω, it will always hold that $\mathit{rank}_\alpha(p, i) \leq \lVert \mathit{rank}_\alpha(q, i) \rVert$.

On the other hand, suppose that $\mathit{rank}_\alpha(q, i)$ is finite, i.e., $\alpha_{i:\omega}$ is not accepted by q. Then, due to Lemma 10.3.1, $0 \leq \mathit{rank}_\alpha(q, i) \leq 2n$. Because $p \preceq_{de} q$, it holds that $\alpha_{i:\omega}$ is also not accepted by p, and therefore also $0 \leq \mathit{rank}_\alpha(p, i) \leq 2n$. We now need to show that $0 \leq \mathit{rank}_\alpha(p, i) \leq \lVert \mathit{rank}_\alpha(q, i) \rVert \leq 2n$.

Let $\{\mathcal{G}_\alpha^k\}_{k=0}^{2n+1}$ be the sequence of run DAGs obtained from $\mathcal{G}_\alpha$ in the ranking procedure from Section 10.3.1. In the following text we use the abbreviation $v \in \mathcal{G}_\alpha^m \setminus \mathcal{G}_\alpha^n$ for $v \in \mathcal{G}_\alpha^m \wedge v \notin \mathcal{G}_\alpha^n$. Since the rank of a node (r, j) is given as the number l s.t. $(r, j) \in \mathcal{G}_\alpha^l \setminus \mathcal{G}_\alpha^{l+1}$, we will finish the proof of this lemma by proving the following claim:

Claim 6: *Let k and l be s.t. $(p, i) \in \mathcal{G}_\alpha^k \setminus \mathcal{G}_\alpha^{k+1}$ and $(q, i) \in \mathcal{G}_\alpha^l \setminus \mathcal{G}_\alpha^{l+1}$. Then $k \leq \lVert l \rVert$.*

Proof We prove the claim by induction on l.

- Base case: $(l = 0)$ Since we assume $\mathcal{A}$ is complete, no vertex in $\mathcal{G}_\alpha^0$ is finite.

  $(l = 1)$ We prove that if (q, i) is endangered in $\mathcal{G}_\alpha^1$, then (p, i) is endangered in $\mathcal{G}_\alpha^1$ as well (so both would be removed in $\mathcal{G}_\alpha^2$). For the sake of contradiction, assume that (q, i) is endangered in $\mathcal{G}_\alpha^1$ and (p, i) is not. Therefore, since $\mathcal{G}_\alpha^1$ contains no finite vertices, there is an infinite path π from (p, i) s.t. π contains at least one accepting state. In the following, we abuse notation and, given a strategy σ_{de} and a state $s \in Q$, use $\sigma_{de}((s, i), \pi)$ to denote the path $(s_0, i)(s_1, i + 1)(s_2, i + 2) \dots$ such that $s_0 = s$ and $\forall j \geq 0$, it holds that $s_{j+1} = \sigma_{de}(s_j, r_{i+j} \xrightarrow{\alpha_{i+j}} r_{i+j+1})$ where $\pi_x = (r_x, x)$ for every $x \geq 0$. Since $p \preceq_{de} q$, there is a corresponding infinite path $\pi' = \sigma_{de}((q, i), \pi)$ that also contains at least one accepting state. Therefore, (q, i) is not endangered, a contradiction to the assumption, so we conclude that $l = 1 \Rightarrow k = 1$.

- Inductive step: We assume the claim holds for all $l < 2j$ and prove the inductive step for even and odd steps independently.

  $(l = 2j)$ We prove that if (q, i) is finite in $\mathcal{G}_\alpha^l$ (and therefore would be removed in $\mathcal{G}_\alpha^{l+1}$), then either $(p, i) \notin \mathcal{G}_\alpha^l$, or (p, i) is also finite in $\mathcal{G}_\alpha^l$. For the sake of contradiction, we assume that (q, i) is finite in $\mathcal{G}_\alpha^l$ and that (p, i) is in $\mathcal{G}_\alpha^l$, but is not finite there (and, therefore, $k > l$). Since (p, i) is not finite in $\mathcal{G}_\alpha^l$, there is an infinite path π from (p, i) in $\mathcal{G}_\alpha^l$. Because $p \preceq_{de} q$, it follows that there is an infinite path $\pi' = \sigma_{de}((q, i), \pi)$ in $\mathcal{G}_\alpha^0$ (π' is not in $\mathcal{G}_\alpha^l$ because (q, i) is finite there). Using Lemma 11.2.4 (possibly multiple times) and the fact that (q, i) is finite, we can find vertices (p', x) in π and (q', x) in π' s.t. $p' \preceq_{de} q'$ and (q', x) is not in $\mathcal{G}_\alpha^l$, therefore, $(q', x) \in \mathcal{G}_\alpha^e \setminus \mathcal{G}_\alpha^{e+1}$ for some $e < l$. Because $(p', x) \in \mathcal{G}_\alpha^l$ and it is not finite (π is infinite), it follows that $(p', x) \in \mathcal{G}_\alpha^f \setminus \mathcal{G}_\alpha^{f+1}$ for some $f > l$, and since $e < l < f$, we have that $f \not\leq e + 1$, implying $f \not\leq \lVert e \rVert$, which is in contradiction to the induction hypothesis.

  $(l = 2j+1)$ We prove that if (q, i) is endangered in $\mathcal{G}_\alpha^l$ (and therefore would be removed in $\mathcal{G}_\alpha^{l+1}$), then either $(p, i) \notin \mathcal{G}_\alpha^l$, or (p, i) is removed at the latest in $\mathcal{G}_\alpha^{l+1}$. For the sake of contradiction, assume that (q, i) is endangered in $\mathcal{G}_\alpha^l$ while (p, i) is removed later than in $\mathcal{G}_\alpha^{l+1}$. Therefore, since $\mathcal{G}_\alpha^l$ contains no finite vertices (they were removed in the $(l - 1)$-th step), there is an infinite path π from (p, i) s.t. π contains at least one accepting state. Because $p \preceq_{de} q$, there is a corresponding path $\pi' = \sigma_{de}((q, i), \pi)$ from (q, i) in $\mathcal{G}_\alpha^0$ that also contains at least one accepting state and moreover $\pi' \notin \mathcal{G}_\alpha^l$. Since π' has an infinite number of states (and at least one accepting), not all states from π' were removed in $\mathcal{G}_\alpha^{l-1}$, i.e., there is at least one node with rank less or equal to $l - 2$. Using Lemma 11.2.4 (also possibly multiple times) we can hence find states (p', x) in π and (q', x) in π' s.t. $p' \preceq_{de} q'$ and (q', x) is not in $\mathcal{G}_\alpha^l$ and has a rank less or equal to $l - 2$, therefore, $(q', x) \in \mathcal{G}_\alpha^e \setminus \mathcal{G}_\alpha^{e+1}$ for some $e < l - 1$. Because $(p', x) \in \mathcal{G}_\alpha^l$, it follows that $(p', x) \in \mathcal{G}_\alpha^f \setminus \mathcal{G}_\alpha^{f+1}$ for some $f \geq l$, and, therefore, $f \not\leq e + 1$, which is in contradiction to the induction hypothesis. ■

This concludes the proof. □

Lemma 11.2.6. *Let $p, q \in Q$ such that $p \preceq_{di} q$ and $\mathcal{G}_\alpha = (V, E)$ be the run DAG of $\mathcal{A}$ over α. For all $i \geq 0$, it holds that $(p, i) \in V \wedge (q, i) \in V \Rightarrow rank_\alpha(p, i) \leq rank_\alpha(q, i)$.*

Proof. Can be obtained as a simplified version of the proof of Lemma 11.2.5. □

We are now ready to prove Lemma 11.2.2.

Lemma 11.2.2. $\mathcal{L}\left(\textsc{Prg}_{de}(\mathcal{A})\right) = \mathcal{L}\left(\textsc{Schewe}(\mathcal{A})\right)$

Proof. ($\subseteq$) Follows directly from the fact that $\textsc{Prg}_{de}(\mathcal{A})$ is obtained by removing states from $\textsc{Schewe}(\mathcal{A})$.

($\supseteq$) Let $\alpha \in \mathcal{L}\left(\textsc{Schewe}(\mathcal{A})\right)$. As shown in the proof of Lemma 3.2 in [250], there are two cases. The first case is when all vertices of $\mathcal{G}_\alpha$ are finite, which we do not need to consider, since we assume complete automata. The other case is when $\mathcal{G}_\alpha$ contains an infinite vertex. In this case, $\textsc{Schewe}(\mathcal{A})$ contains an accepting run

$$\rho = S_0 S_1 \dots S_p (S_{p+1}, O_{p+1}, f_{p+1}, i_{p+1})(S_{p+2}, O_{p+2}, f_{p+2}, i_{p+2}) \dots$$

with

- $S_0 = I, O_{p+1} = \emptyset$, and $i_{p+1} = 0$,
- $S_{j+1} = \delta(S_j, \alpha_j)$ for all $j \in \omega$,

and, for all $j > p$,

- $O_{j+1} = f_{j+1}^{-1}(i_{j+1})$ if $O_j = \emptyset$ or
  $O_{j+1} = \delta(O_j, \alpha_j) \cap f_{j+1}^{-1}(i_{j+1})$ if $O_j \neq \emptyset$, respectively,
- f_j is the S_j-tight level ranking that maps each $q \in S_j$ to the rank of $(q, j) \in \mathcal{G}_\alpha$,
- $i_{j+1} = i_j$ if $O_j \neq \emptyset$ or
  $i_{j+1} = (i_j + 2) \mod (rank(f) + 1)$ if $O_j = \emptyset$, respectively.

The ranks assigned by f_j to states of S_j match the ranks of the corresponding vertices in $\mathcal{G}_\alpha$.

⊛ Using Lemma 11.2.5, we conclude that ρ contains no macrostate (S, O, f, j) where $f(p) > \lVert f(q) \rVert$ and $p \preceq_{de} q$ for $p, q \in S$. Therefore, ρ is also an accepting run in $\textsc{Prg}_{de}(\mathcal{A})$. (We use ⊛ to refer to this paragraph later.) □

Lemma 11.2.1. $\mathcal{L}\left(\textsc{Prg}_{di}(\mathcal{A})\right) = \mathcal{L}\left(\textsc{Schewe}(\mathcal{A})\right)$

Proof. The same as for Lemma 11.2.2 with ⊛ substituted by the following:
⊛ Using Lemma 11.2.6, we conclude that ρ contains no macrostate (S, O, f, j) where $f(p) > f(q)$ and $p \preceq_{di} q$ for $p, q \in S$. So ρ is also an accepting run in $\textsc{Prg}_{di}(\mathcal{A})$. □

Lemma 11.2.3. $\mathcal{L}\left(\textsc{Prg}_{di+de}(\mathcal{A})\right) = \mathcal{L}\left(\textsc{Schewe}(\mathcal{A})\right)$

Proof. The same as for Lemma 11.2.2 with ⊛ substituted by the following:
⊛ Using Lemmas 11.2.6 and 11.2.5, we conclude that ρ contains no macrostate (S, O, f, j) where either $f(p) > f(q)$ and $p \preceq_{di} q$, or $f(p) > \lVert f(q) \rVert$ and $p \preceq_{de} q$ for $p, q \in S$. Therefore, ρ is also an accepting run in $\textsc{Prg}_{di+de}(\mathcal{A})$. □

11.3 Use after Simulation Quotienting

In this short section, we establish that our optimization introduced in Section 11.2 can be applied with no additional cost in the setting when BA complementation is preceded with simulation-based reduction of the input BA (which is usually helpful), i.e., when the simulation is already computed beforehand for another purpose. In particular, we show that simulation-based reduction preserves the simulation (when naturally extended to the quotient automaton).

Given an x-simulation $\preceq_x$ for $x \in \{di, de\}$, we use $\approx_x$ to denote the *x-similarity* relation (i.e., the symmetric fragment) $\approx_x = \preceq_x \cap \preceq_x^{-1}$. Note that since $\preceq_x$ is a preorder, it holds that $\approx_x$ is an equivalence. The *quotient* of a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ w.r.t. $\approx_x$ is defined in the same way as for NFAs (see Section 2.3).

Theorem 11.3.1 ([68], [107])**.** *If $x \in \{di, de\}$, then $\mathcal{L}(\mathcal{A}/\approx_x) = \mathcal{L}(\mathcal{A})$.*

Remark 11.3.1 ([107])**.** $\mathcal{L}(\mathcal{A}/\approx_f) \neq \mathcal{L}(\mathcal{A})$

Finally, the following lemma shows that quotienting preserves direct and delayed simulations, therefore, when complementing $\mathcal{A}$, it is possible to first quotient $\mathcal{A}$ w.r.t. a direct/delayed simulation and then use the same simulation (lifted to the states of the quotient automaton) to optimize the complementation.

Lemma 11.3.1. *Let $\preceq_x$ be the x-simulation on $\mathcal{A}$ for $x \in \{di, de\}$. Then the relation $\preceq_x^{\approx}$ defined as $[q]_x \preceq_x^{\approx} [r]_x$ iff $q \preceq_x r$ is the x-simulation on $\mathcal{A}/\approx_x$.*

Proof. First, we show that $\preceq_x^{\approx}$ is well defined, i.e., if $q \preceq_x r$, then for all $q' \in [q]_x$ and $r' \in [r]_x$, it holds that $q' \preceq_x r'$. Indeed, this holds because $q' \approx_x q$ and $r \approx_x r$, and therefore $q' \preceq_x q \preceq_x r \preceq_x r'$; the transitivity of simulation yields $q' \preceq_x r'$.

Next, let σ_x be a strategy that gives $\preceq_x$. Consider a trace defined as $[\pi_q]_x = [q_0]_x \xrightarrow{\alpha_0} [q_1]_x \xrightarrow{\alpha_1} \cdots$ over a word $\alpha \in \Sigma^\omega$ in $\mathcal{A}/\approx_x$. Then,

(i) for $x = di$ there is a trace $\pi_q = q'_0 \xrightarrow{\alpha_0} q'_1 \xrightarrow{\alpha_1} \cdots$ in $\mathcal{A}$ s.t. $q'_0 \in [q_0]_x$ and $q_i \preceq_x q'_i$ for $i \geq 0$. Therefore, if $[q_i]_x$ is accepting then so is q'_i;

(ii) for $x = de$ there is a trace $\pi_q = q'_0 \xrightarrow{\alpha_0} q'_1 \xrightarrow{\alpha_1} \cdots$ in $\mathcal{A}$ s.t. $q'_0 \in [q_0]_x$, $q_i \preceq_x q'_i$ for $i \geq 0$ and, moreover, if $[q_i]_x$ is accepting then there is q'_k for $k \geq i$ s.t. $q'_k \in F$.

Further, let $[q_0]_x \preceq_x^{\approx} [r_0]_x$. Then there is a trace $\pi_r = \sigma_x(r, \pi_q) = (r = r_0) \xrightarrow{\alpha_0} r_1 \xrightarrow{\alpha_1} \cdots$ simulating π_q in $\mathcal{A}$ from r. Further, consider its projection $[\pi_r]_x = [r_0]_x \xrightarrow{\alpha_0} [r_1]_x \xrightarrow{\alpha_1} \cdots$ into $\mathcal{A}/\approx_x$. For all $i \geq 0$, we have that $q_i \preceq_x r_i$, and therefore also $[q_i]_x \preceq_x^{\approx} [r_i]_x$. Since $\mathcal{C}^x(\pi_q, \pi_r)$, then also $\mathcal{C}^x([\pi_q]_x, [\pi_r]_x)$.

Finally, we show that $\preceq_x^{\approx}$ is maximal. For the sake of contradiction, suppose that $[r]_x$ is x-simulating $[q]_x$ for some $q, r \in Q$ s.t. $q \not\preceq_x r$. Consider a word $\alpha \in \Sigma^\omega$ and a trace $\pi_q = (q = q_0) \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \cdots$ over α in $\mathcal{A}$. Then there is a trace $[\pi_q]_x = [q = q_0]_x \xrightarrow{\alpha_0} [q_1]_x \xrightarrow{\alpha_1} \cdots$ over α in $\mathcal{A}/\approx_x$. According to the assumption, there is also a trace $[\pi_r]_x = [r = r_0]_x \xrightarrow{\alpha_0} [r_1]_x \xrightarrow{\alpha_1} \cdots$ such that $[\pi_r]_x$ is x-simulating $[\pi_q]_x$. But then there will also exist a trace $\pi_r = (r = r_0) \xrightarrow{\alpha_0} r'_1 \xrightarrow{\alpha_1} r'_1 \xrightarrow{\alpha_2} \cdots$ such that $r_i \preceq_x r'_i$ for all $i \in \omega$ and $\mathcal{C}^x(\pi_q, \pi_r)$ (see the previous part of the proof). Therefore, since $\preceq_x$ is maximal, we have that $q \preceq_x r$, which is in contradiction with the assumption. □

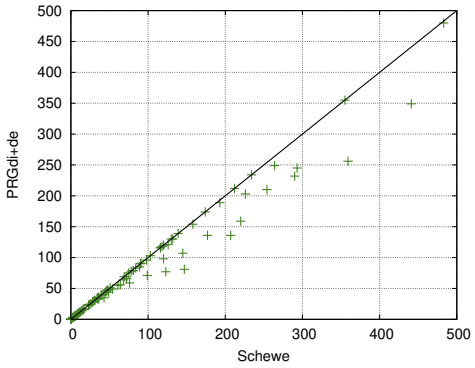


Figure 11.2: PRG_{di+de} vs. SCHEWE

Figure 11.3: Comparison of the number of states of complement BAs generated by SCHEWE and our optimizations (lower is better).

11.4 Experimental Evaluation

We implemented our optimization in a prototype tool[1] written in Haskell and performed preliminary experimental evaluation on a set of 124 random BAs with a non-trivial language over a two-symbol alphabet generated using Tabakov and Vardi's model [265]. The parameters of input automata were set to the following bounds: number of states: 6–7, transition density: 1.2–1.3, and acceptance density: 0.35–0.5. Before complementing, the BAs were quotiented w.r.t. the direct simulation for experiments with PRG_{di} and the delayed simulation for experiments with PRG_{de} and PRG_{di+de}. The timeout was set to 300 s.

We present the results for our strongest optimization for *outputs* of the size up to 500 states in Figure 11.2. As can be seen in this figure, purging often significantly reduces the size of the output. For outputs of a larger size (we had 11 of them), the results follow a similar trend. For some concrete results, for one BA, the size of the output BA decreased from 4065 (SCHEWE) to 985 (PRG_{di+de}), which yields a reduction to 24 %! Further, we observed that all PRG_x methods usually give similar results, with the difference of only a few states (when PRG_{di} and PRG_{de} differ, PRG_{di} usually wins over PRG_{de}).

11.5 Conclusion

We developed a novel optimization of the rank-based complementation algorithm for Büchi automata that is based on leveraging direct and delayed simulation relations to reduce the number of states of the complemented automaton. The optimization is directly usable in rank-based BA inclusion and universality checking. We conjecture that the decision problem of checking BA language inclusion might also bring another opportunities for exploiting simulation, such as in a similar manner as in [14]. Another, orthogonal, directions of future work are (i) applying simulation in other than the rank-based approach (in addition to the particular use within [12, 13]), e.g., complementation based on Safra's construction [248], which, according to our experience, often produces smaller complements than the rank-based procedure, (ii) applying our ideas within determinization constructions for BAs, and (iii) generalizing our techniques for richer simulations, such as the multi-pebble

[1] `https://github.com/vhavlena/ba-complement`

simulation [108] or various look-ahead simulations [206, 207]. Since the richer simulations are usually harder to compute, it would be interesting to find the sweet spot between the overhead of simulation computation and the achieved state space reduction. The content of this chapter was published in the proceedings of APLAS'19 [75].

Chapter 12

Efficient Rank-based Complementation

In Chapter 11, we focused on optimizations of Schewe's construction. We employed direct and delay simulation between states of the original automaton to remove states from Schewe's complement that are not necessary for accepting a word. In this chapter, we build upon these results and develop novel optimizations for reducing the size of the complement that push the rank-based approach by a significant step further.

Recall that Schewe's construction stores in a macrostate partial information about all runs over some word in an input BA. In order to track information about all runs, a macrostate contains a set of states representing a single level in a run DAG of some word with a number assigned to each state representing its rank. The number of macrostates (and hence the size of the complemented automaton) is combinatorially related to the maximum rank that occurs in macrostates (see Section 10.3 for an introduction to rank-based complementation including Schewe's construction).

The theoretical upper bound of the maximal rank for a given macrostate is often too coarse and hence a lot of unnecessary states are generated during the complementation construction. In this chapter, we propose novel optimizations that (among others) reduce this maximum considered rank. We build on a novel notion of a *super-tight run*, i.e., a run in the complement that uses as small ranks as possible. Based on the notion of super-tight runs, we propose a series of optimizations reducing the number of generated states/transitions in Schewe's construction of a complemented automaton with promising experimental results.

Overview of the proposed approach. Our optimizations reason about super-tight runs allowing us to remove states not occurring in any super-tight run from the automaton without affecting its language. Further, based on super-tight runs, we can reduce the maximum rank within a macrostate. In particular, we reduce the maximum considered rank by reasoning about the deterministic support of an input automaton or by a relation based on direct simulation that imply rank ordering computed *a priori* from the input automaton. We introduce also an optimization keeping in the complemented automaton only significant runs. The developed optimizations give, to the best of our knowledge, a very competitive BA complementation procedure as witnessed by our experimental evaluation.

These optimizations require some additional computational cost, but from the perspective of BA complementation, their cost is still negligible and, as we show in our experimental

evaluation, their effect on the size of the output is often profound, in many cases by one or more orders of magnitude. Rank-based complementation with our optimizations is now competitive with other complementation approaches. On the considered benchmark of hard instances, in a large number of cases (21 %) we obtained a strictly smaller complement than *any other existing tool* and in the majority of cases (63 %) we obtained an automaton at least as small as the smallest automaton provided by any other tool.

Related work. The problem of BA complementation was discussed in Chapter 10. We recall only the most relevant works to this chapter (mentioned also in Chapter 11). The work in [133] contains optimizations of an alternative (sub-optimal) rank-based construction from [182] that goes through alternating Büchi automata. Furthermore, the work in [163] proposes an optimization of Schewe that in some cases produces smaller automata (the construction is not compatible with our optimizations).

Chapter outline. This chapter is organized as follows. Section 12.1 introduces super-tight runs. In Section 12.2, we propose our optimizations reducing the generated state space. Section 12.3 then focuses on experimental evaluation of the implementation. Finally, Section 12.4 concludes the chapter.

12.1 Super-tight Runs

Before we move to the proposed optimizations, we introduce a notion of super-tight runs, which is a fundamental stone of our improved construction presented later. In this chapter, we assume definitions from Chapters 2, 10, and 11. We extend them with definitions used in this chapter. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a BA. We use $\delta^{-1}(q, a)$ to denote the set $\{s \in Q \mid s \xrightarrow{a} q \in \delta\}$. For a set of states S, we define *reachability* from S as $\mathit{reach}_\delta(S) = \mu Z.\, S \cup \bigcup_{a \in \Sigma} \delta(Z, a)$. In this chapter, we fix a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$.

In the following, we assume that Schewe($\mathcal{A}$) (see Section 10.3.4) contains only the states and transitions reachable from I'. We begin our optimizations with a notion of super-tight runs. We use Schewe as the basis for further optimizations in the rest of the chapter. Let $\mathcal{B} =$ Schewe($\mathcal{A}$). Each accepting run of $\mathcal{B}$ on $\alpha \in \mathcal{L}(\mathcal{B})$ is *tight*, i.e., the rankings of macrostates it traverses in Q_2 are tight (this follows from the definition of Q_2). In this section, we show that there exists a *super-tight run* of $\mathcal{B}$ on α, which is, intuitively, a run that uses as little ranks as possible. Our optimizations in Section 12.2 are based on preserving super-tight runs of $\mathcal{B}$.

Let $\rho = S_0 \dots S_m (S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})(S_{m+2}, O_{m+2}, f_{m+2}, i_{m+2}) \dots$ be an accepting run of $\mathcal{B}$ over a word $\alpha \in \Sigma^\omega$. Given a macrostate (S_k, O_k, f_k, i_k) for $k > m$, we define its *rank* as $\mathit{rank}((S_k, O_k, f_k, i_k)) = \mathit{rank}(f_k)$. Further, we define the *rank of the run* ρ as $\mathit{rank}(\rho) = \min\{\mathit{rank}((S_k, O_k, f_k, i_k)) \mid k > m\}$.

Let $\mathcal{G}_\alpha$ be the run DAG of $\mathcal{A}$ over α and rank_α be the ranking of vertices in $\mathcal{G}_\alpha$. We say that the run ρ is *super-tight* if for all $k > m$ and all $q \in S_k$, it holds that $f_k(q) = \mathit{rank}_\alpha(q, k)$. Intuitively, super-tight runs correspond to runs whose ranking faithfully copies the ranks assigned in $\mathcal{G}_\alpha$ (from some position m corresponding to the transition from the waiting to the tight part of $\mathcal{B}$).

Lemma 12.1.1. *Let $\alpha \in \mathcal{L}(\mathcal{B})$. Then there is a super-tight accepting run ρ of $\mathcal{B}$ on α.*

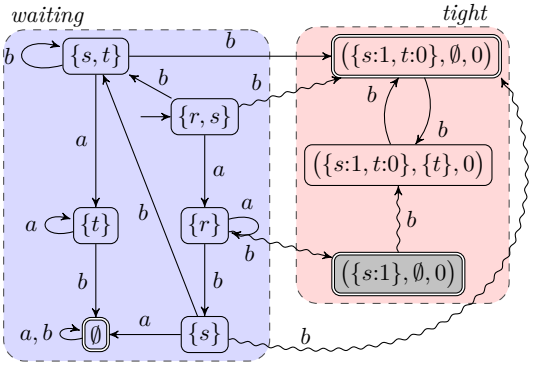


Figure 12.1: Illustration of DELAY.

Proof. This follows directly from the definition of a super-tight run and the SCHEWE construction. □

Let $\rho = S_0 \dots S_m(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})(S_{m+2}, O_{m+2}, f_{m+2}, i_{m+2}) \dots$ be a run and consider a macrostate (S_k, O_k, f_k, i_k) for $k > m$. We call a set $C_k \subseteq S_k$ a *tight core of a ranking* f_k if $f_k(C_k) = \{1, 3, \dots, rank(f_k)\}$ and $f_k|_{C_k}$ is injective (i.e., every state in the tight core has a unique odd rank). Moreover, C_k is a *tight core of a macrostate* (S_k, O_k, f_k, i_k) if it is a tight core of f_k. We say that an infinite sequence $\tau = C_{m+1}C_{m+2} \dots$ is a *trunk* of run ρ if for all $k > m$ it holds that C_k is a tight core of $\rho(k)$ and there is a bijection $\theta : C_k \to C_{k+1}$ s.t. if $\theta(q_k) = q_{k+1}$ then $q_{k+1} \in \delta(q_k, \alpha_k)$. We will, in particular, be interested in trunks of super-tight runs. In these runs, a trunk (there can be several) *represents runs of $\mathcal{A}$ that keep the super-tight ranks of ρ*. The following lemma shows that every state in any tight core in a trunk of such a run has at least one successor with the same rank.

Lemma 12.1.2. *Let $\rho = S_0 \dots S_m(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1}) \dots$ be an accepting super-tight run of $\mathcal{B}$ on α. Then there is a trunk $\tau = C_{m+1}C_{m+2} \dots$ of ρ and, moreover, for every $k > m$ and all states $q_k \in C_k$, it holds that there is a state $q_{k+1} \in C_{k+1}$ such that $f_k(q_k) = f_{k+1}(q_{k+1})$.*

Proof. First we show how to inductively construct a trunk $\tau = C_{m+1}C_{m+2} \dots$ (i) As the base case, let C_{m+1} be an arbitrary tight core of f_{m+1}. (ii) As the inductive step, consider a tight core C_k from the trunk and let us construct C_{k+1}. Since ρ is a super-tight run, for each $q \in C_k$ there is a state $q' \in S_{k+1}$ s.t. $f_k(q) = f_{k+1}(q')$. This follows from the run DAG ranking procedure. We put $q' \in C_{k+1}$. Such a constructed set is a tight core of f_{k+1} and from the construction we have the property stated in the lemma. □

12.2 Optimized Complement Construction

In this section, we introduce our optimizations of SCHEWE that are key to producing small complement automata in practice.

12.2.1 Delaying the Transition from Waiting to Tight

Our first optimization of the construction of the complement automaton reduces the number of nondeterministic transitions between the *waiting* and the *tight* part. This optimization

Algorithm 5: The DELAY construction

Input: A Büchi automaton $\mathcal{A} = (Q, \Sigma, I, \delta, F)$
Output: A Büchi automaton $\mathcal{C}$ s.t. $\mathcal{L}(\mathcal{C}) = \overline{\mathcal{L}(\mathcal{A})}$

$\mathcal{S} \leftarrow \{I\}$, $Q_1 \leftarrow \{I\}$, $\theta_2 \leftarrow \emptyset$, $(\cdot, \Sigma, \delta_1 \cup \delta_2 \cup \delta_3, I', F') \leftarrow$ SCHEWE($\mathcal{A}$);
while $\mathcal{S} \neq \emptyset$ **do**
  Take a waiting-part macrostate $R \subseteq Q$ from $\mathcal{S}$;
  foreach $a \in \Sigma$ **do**
    if $\exists T \in \delta_1(R, a)$ *s.t.* $R \xrightarrow{a} T$ *closes a cycle in* Q_1 **then**
      $\theta_2 \leftarrow \theta_2 \cup \{R \xrightarrow{a} U \mid U \in \delta_2(R, a)\}$;
    foreach $T \in \delta_1(R, a)$ *s.t.* $T \notin Q_1$ **do**
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{T\}$;
      $Q_1 \leftarrow \mathcal{Q} \cup \{T\}$;
$Q_2 \leftarrow \mathit{reach}_{\delta_3}(\mathrm{img}(\theta_2))$;
return $\mathcal{C} = (Q_1 \cup Q_2, \Sigma, \delta_1 \cup \theta_2 \cup \delta_3, I', F' \cap Q_2)$;

is inspired by the idea of *partial order reduction* in model checking [126, 279, 226]. In particular, since in each state of the waiting part, it is possible to move to the tight part, we can arbitrarily delay such a transition (but need to take it eventually) and, therefore, significantly reduce the number of transitions (and, as our experiments later show, also significantly reduce the number of reachable states in Q_2).

Speaking in the terms of partial order reduction, when constructing the waiting part of the complement BA, given a macrostate $S \in Q_1$ and a symbol $a \in \Sigma$, we can set $\theta_2 \subseteq \delta_2$ such that $\theta_2(S, a) := \emptyset$ if the *cycle closing condition* holds and $\theta_2(S, a) := \delta_2(S, a)$ otherwise. Informally, the *cycle closing condition* (often denoted as **C3**) holds for S and a if the successor of S over a in the waiting part does not close a cycle where the transition to the tight part would be infinitely often delayed. Practically, it means that when constructing Q_1, we need to check whether successors of a macrostate close a cycle in the so-far generated part of Q_1. We give the construction in Algorithm 5 and we refer to it as DELAY. Using this optimization on the example in Figure 12.1, we would remove the b-transitions from $\{r, s\}$ and $\{s\}$ to the macrostate $(\{s{:}1, t{:}0\}, \emptyset, 0)$ and also the macrostate $(\{s{:}1\}, \emptyset, 0)$ (including the transitions incident with it).

Lemma 12.2.1. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}(\textsc{Delay}(\mathcal{A})) = \mathcal{L}(\textsc{Schewe}(\mathcal{A}))$. Moreover, for every accepting super-tight run of $\textsc{Schewe}(\mathcal{A})$ on α, there is an accepting super-tight run of $\textsc{Delay}(\mathcal{A})$ on α.*

Proof. Showing $\mathcal{L}(\textsc{Delay}(\mathcal{A})) \subseteq \mathcal{L}(\textsc{Schewe}(\mathcal{A}))$ is trivial. In order to show the reverse direction, consider some $\alpha \in \mathcal{L}(\textsc{Schewe}(\mathcal{A}))$. Then, there is an accepting run $\rho_m = S_0 \dots S_m(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1}) \dots$ on α in $\textsc{Schewe}(\mathcal{A})$. For each $\ell > m$ there is, however, also an accepting run $\rho_\ell = S_0 \dots S_\ell(S_{\ell+1}, O_{\ell+1}, f_{\ell+1}, i_{\ell+1}) \dots$ on α in $\textsc{Schewe}(\mathcal{A})$. Note that each ρ_ℓ differs from ρ_m on the point where the run switched from the waiting part to the tight part of $\textsc{Schewe}(\mathcal{A})$. Therefore, since the run ρ_m managed to empty O infinitely often, ρ_ℓ will also be able to do so and, therefore, it will also be accepting.

From the properties of DELAY construction, we have that at least one macrostate $(S_{k+1}, O_{k+1}, f_{k+1}, i_{k+1})$ where $k > m$ is in Q_2. If this were not true, there would be a closed cycle with no state in Q_2, which is a contradiction. From the previous reasoning we have that the run $\rho_k = S_1 \dots S_k(S_{k+1}, O_{k+1}, f_{k+1}, i'_{k+1}) \dots$ on α is present in $\textsc{Delay}(\mathcal{A})$.

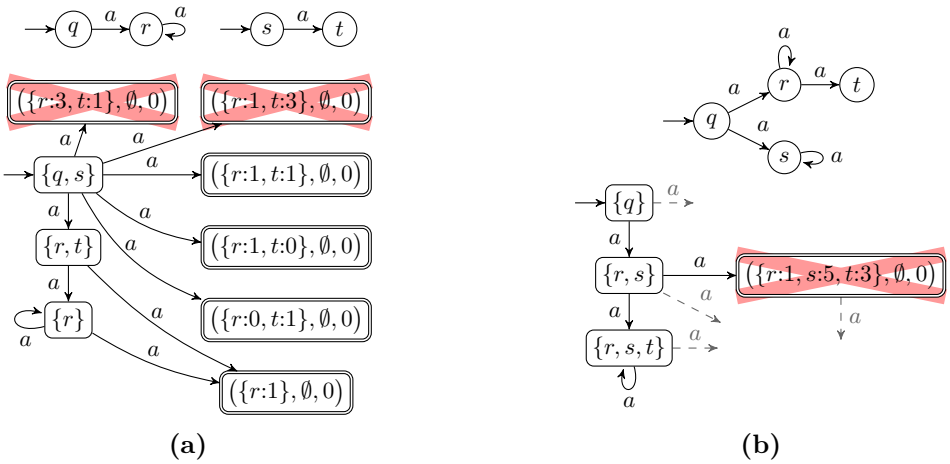


Figure 12.2: Illustration of SuccRank reduction (φ_{coarse}), focusing on the transitions from the waiting to the tight part (a), and SuccRank reduction (φ_{fine}), focusing on one particular macrostate (b).

Moreover, this run is accepting both in Schewe($\mathcal{A}$) and Delay($\mathcal{A}$), which concludes the proof. □

Since Delay does not affect the rankings in the macrostates and only delays the transition from the waiting to the tight part, we can freely use it as the base algorithm instead of Schewe in all following optimizations.

12.2.2 Successor Rankings

Our next optimization is used to reduce the maximum considered ranking of a macrostate in the tight part of $\mathcal{B} = \text{Schewe}(\mathcal{A})$. For a given macrostate, the number of tight rankings that can occur within the macrostate rises combinatorially with the macrostate's maximum rank (in particular, the number of tight rankings for a given set of states corresponds to the Stirling number of the second kind of the maximum rank [115]). It is hence desirable to reduce the maximum considered rank as much as possible.

The idea of our optimization called SuccRank is the following. Suppose we have a macrostate (S, O, f, i) from the tight part of $\mathcal{B}$. Further, assume that the maximum number of non-accepting states in the S-component of a macrostate that is infinitely often reachable from (S, O, f, i) is $\lceil S \rceil$. Then, we know that a super-tight accepting run that goes through (S, O, f, i) will never need a rank higher than $2\lceil S \rceil - 1$ (any accepting state will be assigned an even rank, so we can omit them). Therefore, if the rank of f is higher than $2\lceil S \rceil - 1$, we can safely discard (S, O, f, i) (since there will be a super-tight accepting run that goes over (S, O', f', i') with $f' < f$). This part of the optimization is called *coarse*.

Moreover, let $q \in S$ and let $\lfloor \{q\} \rfloor$ be the smallest size of a set of states (again without accepting states) reachable from q over some (infinite) word infinitely often. Then, we know that those states will have a rank bounded by the rank of $f(q)$, so there are only (at most) $\lceil S \rceil - \lfloor \{q\} \rfloor$ states whose rank can be higher than $f(q)$. Therefore, the rank of f, which is tight, can be at most $f(q) + 2(\lceil S \rceil - \lfloor \{q\} \rfloor)$. We call this part of the optimization *fine*.

We now formalize the intuition. Let us fix a BA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$. Then, let us consider a BA $R_{\mathcal{A}} = (2^Q, \Sigma, \delta_R, \emptyset, \emptyset)$, with $\delta_R = \{R \xrightarrow{a} S \mid S = \delta(R, a)\}$, which is tracking *reachability* between set of all states of $\mathcal{A}$ (we only focus on its structure and not the

language). Note that $R_{\mathcal{A}}$ is deterministic and complete. Further, given $S \subseteq Q$, let us use $SCC(S) \subseteq 2^{2^Q}$ to denote the set of all *strongly connected components reachable from S in $R_{\mathcal{A}}$*. We will use *inf-reach*(S) to denote the set of states $\bigcup SCC(S)$, i.e., the set of states such that there is an infinite path in $R_{\mathcal{A}}$ starting in S that passes through a given state infinitely many times.

For $S \subseteq Q$, we define the maximum and minimum sizes of macrostates reachable infinitely often from S:

$$\lceil S \rceil = \max\{|R \setminus F| \mid R \in \textit{inf-reach}(S)\} \quad \text{and} \tag{12.1}$$

$$\lfloor S \rfloor = \min\{|R \setminus F| \mid R \in \textit{inf-reach}(S)\}. \tag{12.2}$$

Given a macrostate (S, O, f, i), we define the condition

$$\varphi_{\textit{coarse}}((S, O, f, i)) \stackrel{\text{def}}{\equiv} \textit{rank}(f) \leq 2\lceil S \rceil - 1. \tag{12.3}$$

If the macrostate (S, O, f, i) does not satisfy $\varphi_{\textit{coarse}}$, we do not need to include it in the output of SCHEWE($\mathcal{A}$) (as allowed by Lemma 12.2.2). See Figure 12.2a for an example of macrostates not satisfying $\varphi_{\textit{coarse}}$. For instance, macrostate $(\{r{:}3, t{:}1\}, \emptyset, 0)$ can be removed since its rank is 3 and $\lceil \{r, t\} \rceil = 1$, so $3 \not\leq 2\lceil \{r, t\} \rceil - 1$.

Moreover, we also define the condition

$$\varphi_{\textit{fine}}((S, O, f, i)) \stackrel{\text{def}}{\equiv} \textit{rank}(f) \leq \min\{f(q) + 2(\lceil S \rceil - \lfloor \{q\} \rfloor) \mid q \in S\}. \tag{12.4}$$

Again, if (S, O, f, i) does not satisfy $\varphi_{\textit{fine}}$, it does not need to be in the result. See Figure 12.2b for an example of such a macrostate. Note that the rank of $(\{r{:}1, s{:}5, t{:}3\}, \emptyset, 0)$ is 5, $\lceil \{r, s, t\} \rceil = 3$ and $\lfloor \{r\} \rfloor = 2$, $\lfloor \{s\} \rfloor = 1$, $\lfloor \{t\} \rfloor = 0$. Then, $\min\{f(r) + 2(3-2), f(s) + 2(3-1), f(t) + 2(3-0)\} = \min\{1+2, 5+4, 3+6\} = 3$, so $5 \not\leq 3$ and the macrostate does not satisfy $\varphi_{\textit{fine}}$ and can be removed.

We emphasize that $\varphi_{\textit{coarse}}$ and $\varphi_{\textit{fine}}$ are incomparable. For example, the macrostates removed due to $\varphi_{\textit{coarse}}$ in Figure 12.2a satisfy $\varphi_{\textit{fine}}$ (since, e.g., $3 \leq \min\{3 + 2(1-1), 1 + 2(1-0)\}$) and the macrostate removed due to $\varphi_{\textit{fine}}$ in Figure 12.2b satisfies $\varphi_{\textit{coarse}}$ (since $5 \leq 2 \cdot 3 - 1$).

Putting the conditions together, we define the predicate

$$\textsc{SuccRank}((S, O, f, i)) \stackrel{\text{def}}{\equiv} \varphi_{\textit{coarse}}((S, O, f, i)) \wedge \varphi_{\textit{fine}}((S, O, f, i)). \tag{12.5}$$

We abuse notation and use SUCCRANK($\mathcal{A}$) to denote the output of SCHEWE($\mathcal{A}$) where the states from the tight part of SCHEWE($\mathcal{A}$) are restricted to those that satisfy SUCCRANK.

Lemma 12.2.2. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}$ (SUCCRANK($\mathcal{A}$)) = $\mathcal{L}$ (SCHEWE($\mathcal{A}$)).*

Proof. The inclusion $\mathcal{L}$ (SUCCRANK($\mathcal{A}$)) $\subseteq$ $\mathcal{L}$ (SCHEWE($\mathcal{A}$)) is clear. Now we look at the other direction. Consider some $\alpha \in \mathcal{L}$ (SCHEWE($\mathcal{A}$)). Then, there is an accepting supertight run $\rho = S_0 \dots S_m (S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1}) \dots$ of SCHEWE($\mathcal{A}$) over α. Consider $k > m$ and a macrostate (S_k, O_k, f_k, i_k). The maximum rank of this macrostate is bounded by $2\lceil S_k \rceil - 1$ because $\lceil S_k \rceil$ is the largest size of the S-component (without final states) of a macrostate reachable from S_k and, therefore, removing macrostates that do not satisfy $\varphi_{\textit{coarse}}$ from SCHEWE($\mathcal{A}$) will not affect this run.

Next, we prove the correctness of removing states from SCHEWE($\mathcal{A}$) using $\varphi_{\textit{fine}}$. Consider a set of states $T \subseteq Q$; we will use ρ_T to denote the run $\rho_T = T_0 T_1 T_2 \dots$ of $R_{\mathcal{A}}$ from T

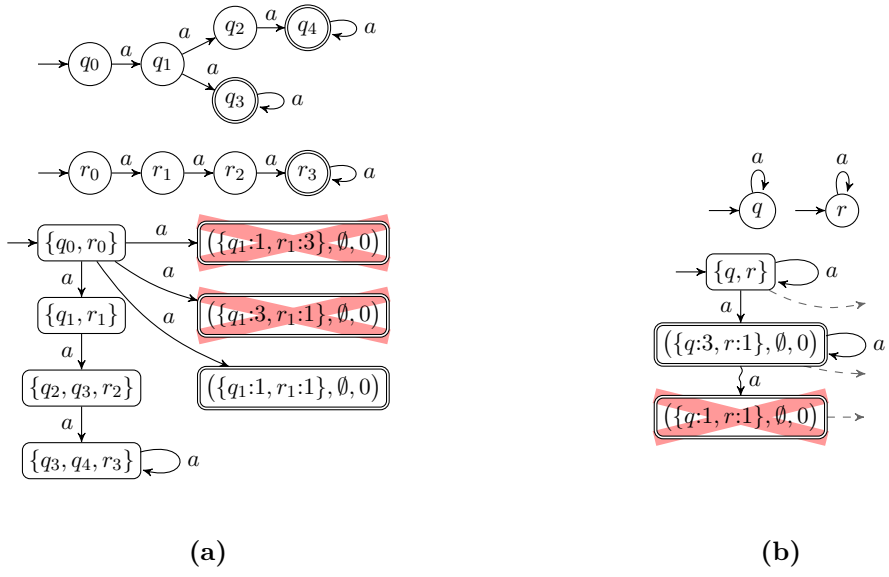


Figure 12.3: Illustration of RankSim′ (a) and RankRestr (b).

($= T_0$) over the word $\alpha_{k:\omega}$. Since $R_{\mathcal{A}}$ is deterministic and complete, there is exactly one such run. Given a state $q \in S_k$, let a be the smallest size of a set of states (without final states) that occurs infinitely often in $\rho_{\{q\}}$ and b be the largest size of a set of states that occurs infinitely often in ρ_{S_k} (again without final states). From the definition of $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$, it holds that

$$\lfloor \{q\} \rfloor \;\leq\; a \;\leq\; b \;\leq\; \lceil S_k \rceil. \tag{12.6}$$

Since we can reach a different states from q, the ranks of these states need to be less or equal to $f_k(q)$ (no successor of q can be given a rank higher than $f_k(q)$). Further, since we can reach at most b different states from S_k, there are at some infinitely often occurring macrostate of ρ in the worst case only $b - a$ states that can have and odd rank greater than $f_k(q)$. Due to the tightness of all macrostates in the tight part of ρ, we can conclude that the maximal rank of f_k can be bounded by $f_k(q) + 2(\lceil S_k \rceil - \lfloor \{q\} \rfloor)$. Therefore, a macrostate where φ_{fine} does not hold will not be in a super-tight run, so removing those macrostates does not affect the language of Schewe($\mathcal{A}$).

□

12.2.3 Rank Simulation

The next optimization is a modification of optimization Prg$_{di}$ from Chapter 11. Intuitively, Prg$_{di}$ is based on the fact that if a state p is directly simulated by a state r, i.e., $p \preceq_{di} r$, then any macrostate (S, O, f, i) where $f(p) > f(r)$ can be safely removed (intuitively, any run from p can be simulated by a run from r, where the run from r may contain more accepting states and, therefore, needs to decrease its rank more times). Prg$_{di}$ is compatible with Schewe but, unfortunately, it is incompatible with the MaxRank construction (one of our further optimizations introduced in Section 12.2.5) since in MaxRank, several runs are represented by one *maximal* run (w.r.t. the ranks) and removing such a run would also remove the smaller runs (see Section 12.2.5 for details). We, however, change the

condition and obtain a new reduction, which is incomparable with PRG_{di} but compatible with MaxRank. We call this reduction RankSim.

Consider the following relation of *odd-rank simulation* on $p, r \in Q$:

$$\begin{aligned} p \preceq_{ors} r \stackrel{\text{def}}{\equiv} \forall \alpha \in \Sigma^\omega, \forall i \geq 0 : (\mathit{rank}_\alpha(p,i) \text{ is odd} \wedge \mathit{rank}_\alpha(r,i) \text{ is odd}) \\ \Rightarrow \mathit{rank}_\alpha(p,i) \leq \mathit{rank}_\alpha(r,i). \end{aligned} \tag{12.7}$$

Intuitively, if $p \preceq_{ors} r$ holds, then we know that in any super-tight run and a macrostate (S, O, f, i) in such a run, if $p, r \in S$ and both $f(p)$ and $f(r)$ are odd, then it needs to hold that $f(p) \leq f(r)$. Furthermore, such a reasoning can also be applied transitively ($\preceq_{ors}$ is by itself not transitive): if, in addition, $t \in S$, the rank $f(t)$ is odd, and $r \preceq_{ors} t$, then it also needs to hold that $f(p) \leq f(t)$.

Formally, given a ranking f, let $\preceq^f_{ors}$ be a modification of $\preceq_{ors}$ defined as

$$p \preceq^f_{ors} r \stackrel{\text{def}}{\equiv} f(p) \text{ is odd} \wedge f(r) \text{ is odd} \wedge p \preceq_{ors} r \tag{12.8}$$

and $\preceq^{fT}_{ors}$ be its transitive closure. We use $\preceq^{fT}_{ors}$ to define the following condition:

$$\text{RankSim}((S, O, f, i)) \stackrel{\text{def}}{\equiv} \forall p, r \in S \colon p \preceq^{fT}_{ors} r \Rightarrow f(p) \leq f(r). \tag{12.9}$$

Abusing the notation, we use $\text{RankSim}(\mathcal{A})$ to denote the output of $\text{Schewe}(\mathcal{A})$ where states from the tight part of $\text{Schewe}(\mathcal{A})$ are restricted to those that satisfy RankSim.

Lemma 12.2.3. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}(\text{RankSim}(\mathcal{A})) = \mathcal{L}(\text{Schewe}(\mathcal{A}))$.*

Proof. The inclusion $\mathcal{L}(\text{RankSim}(\mathcal{A})) \subseteq \mathcal{L}(\text{Schewe}(\mathcal{A}))$ is clear. For the reverse direction, consider some $\alpha \in \mathcal{L}(\text{Schewe}(\mathcal{A}))$. Then, for α there is a super-tight run $\rho = S_1 \dots S_m (S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1}) \dots$, i.e., for each $k > m$ and each $q \in S_k$ we have $f_k(q) = \mathit{rank}_\alpha(q, k)$. Clearly, each macrostate of ρ satisfies RankSim. □

From the definition of $\preceq_{ors}$, it is not immediate how to compute it, since it is defined over all infinite runs of $\mathcal{A}$ over all infinite words. The computation of a rich under-approximation of $\preceq_{ors}$ will be the topic of the rest of this section. We first note that $\preceq_{di} \subseteq \preceq_{ors}$, which is a consequence of Lemma 11.2.6 (Section 11.2). We will extend $\preceq_{di}$ into a relation $\preceq_R$, which is computed statically on $\mathcal{A}$, and then show that $\preceq_R \subseteq \preceq_{ors}$. The relation $\preceq_R$ is defined recursively as the smallest binary relation over Q such that

(i) $\preceq_{di} \subseteq \preceq_R$ and

(ii) for $p, r \in Q$, if $\forall a \in \Sigma : (\delta(p,a) \setminus F) \preceq^{\forall\forall}_R (\delta(r,a) \setminus F)$, then $p \preceq_R r$.

Above, $S_1 \preceq^{\forall\forall}_R S_2$ holds iff $\forall x \in S_1, \forall y \in S_2 : x \preceq_R y$. The relation $\preceq_R$ can then be computed using a standard *worklist* algorithm, starting from $\preceq_{di}$ and adding pairs of states for which condition 2 holds until a fixpoint is reached.

Lemma 12.2.4. *We have $\preceq_R \subseteq \preceq_{ors}$.*

Proof. The base case $\preceq_{di} \subseteq \preceq_{ors}$ follows directly from Lemma 11.2.6. For the induction step, let $p, r \in Q$ be such that $\forall a \in \Sigma : (\delta(p,a) \setminus F) \preceq^{\forall\forall}_R (\delta(r,a) \setminus F)$. Our induction hypothesis is that for every $a \in \Sigma, x \in (\delta(p,a) \setminus F)$, and $y \in (\delta(r,a) \setminus F)$, it holds that for all $\alpha \in \Sigma^\omega$ and for all $i \geq 0$, if $\mathit{rank}_\alpha(p,i)$ is odd and $\mathit{rank}_\alpha(r,i)$ is odd, then $\mathit{rank}_\alpha(p,i) \leq$

$rank_\alpha(r,i)$. Let us fix an $a \in \Sigma$ and a word $\alpha \in \Sigma^\omega$ that has a at its i-th position. If $rank_\alpha(p,i)$ or $rank_\alpha(r,i)$ are even, the condition holds trivially.

Assume now that $rank_\alpha(p,i)$ and $rank_\alpha(r,i)$ are odd. From the construction of the run DAG $\mathcal{G}_\alpha$ in Section 10.3.1, it follows that there exist infinite paths from (p,i) and (r,i) in $\mathcal{G}_\alpha$ such that all vertices on these paths are assigned the same (odd) ranks as (p,i) and (r,i), respectively. In particular, there are direct successors $(p', i+1)$ of (p,i) and $(r', i+1)$ of (r,i) whose ranks match the ranks of their predecessors. From the induction hypothesis, it holds that $rank_\alpha(p', i+1) \leq rank_\alpha(r', i+1)$ and so $rank_\alpha(p,i) \leq rank_\alpha(r,i)$ and the lemma follows. (Note that in the previous reasoning, it is essential that (p,i) and (r,i) have an odd ranking; if a node has an even ranking in $\mathcal{G}_\alpha$, then the condition that there needs to be a successor with the same ranking does not hold in general.) □

Putting it all together, we modify (12.9) by substituting $\preceq_{ors}^{fT}$ with $\preceq_R^{fT}$, which denotes the transitive closure of $\preceq_R^f$, where $\preceq_R^f$ is a relation defined (by modifying (12.8)) as

$$p \preceq_R^f r \stackrel{\text{def}}{\equiv} f(p) \text{ is odd} \wedge f(r) \text{ is odd} \wedge p \preceq_R r. \tag{12.10}$$

Because $\preceq_R \subseteq \preceq_{ors}$, Lemma 12.2.3 still holds. We denote the modification of RankSim that uses $\preceq_R^{fT}$ instead of $\preceq_{ors}^{fT}$ as RankSim′.

Example 12.2.1. *Consider the BA $\mathcal{A}$ (top) and the part of Schewe($\mathcal{A}$) (bottom) in Figure 12.3a. Note that $r_2 \preceq_{di} q_2$ and $q_2 \preceq_{di} r_2$ so $r_2 \preceq_R q_2$ and $q_2 \preceq_R r_2$. From the definition of $\preceq_R$, we can deduce that $r_1 \preceq_R q_1$ (since $\{r_2\} \preceq_R^{\forall\forall} \{q_2\}$) and $q_1 \preceq_R r_1$ (since $\{q_2\} \preceq_R^{\forall\forall} \{r_2\}$). Note that $q_1 \not\preceq_{di} r_1$). As a consequence and due to the odd ranks of q_1 and r_1, we can eliminate the macrostates $(\{q_1{:}1, r_1{:}3\}, \emptyset, 0)$ and $(\{q_1{:}3, r_1{:}1\}, \emptyset, 0)$.*

12.2.4 Ranking Restriction

Another optimization, called RankRestr, restricts ranks of successors of states with an odd rank. In particular, in a super-tight run, every odd-ranked state has a successor with the same rank (this follows from the construction of the run DAG). Let $\mathcal{A}$ be a BA and $\mathcal{B} = \text{Schewe}(\mathcal{A}) = (Q, \Sigma, \delta_1 \cup \delta_2 \cup \delta_3, I, F)$. Then, we define the following restriction on transitions:

$$\begin{aligned} \text{RankRestr}((S,O,f,i) \xrightarrow{a} (S',O',f',i')) \stackrel{\text{def}}{\equiv} \\ \forall q \in S : f(q) \text{ is odd } \Rightarrow (\exists q' \in \delta(q,a) : f'(q') = f(q)). \end{aligned} \tag{12.11}$$

We abuse notation and use RankRestr($\mathcal{A}$) to denote $\mathcal{B}$ with transitions from δ_3 restricted to those that satisfy RankRestr. See Figure 12.3b for an example of a transition (and a newly unreachable macrostate) removed using RankRestr.

Lemma 12.2.5. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}(\text{RankRestr}(\mathcal{A})) = \mathcal{L}(\text{Schewe}(\mathcal{A}))$.*

Proof. The inclusion $\mathcal{L}(\text{RankRestr}(\mathcal{A})) \subseteq \mathcal{L}(\text{Schewe}(\mathcal{A}))$ is clear. We now focus on the reverse direction. For that, consider a word $\alpha \in \mathcal{L}(\text{Schewe}(\mathcal{A}))$. Further, let $\rho = S_0 \dots S_m(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1}) \dots$ be an accepting super-tight run on α. Now consider a macrostate (S_j, O_j, f_j, i_j) where $j > m$ and some state $q \in S_j$. Since ρ is super-tight, i.e., represents the run DAG of $\mathcal{A}$ over α, it holds that if $f_j(q)$ is odd, then there is a state $q' \in S_{j+1}$ s.t. $f_{j+1}(q') = f_j(q)$, which satisfies RankRestr. □

12.2.5 Maximum Rank Construction

Our next optimization, named MaxRank, is the one with the biggest practical effect. We introduce it as the last one because it depends on our previous optimizations (in particular SuccRank and RankSim′). It is a modified version of Schewe's "Reduced Average Outdegree" construction [250, Section 4], named Schewe$_{\text{RedAvgOut}}$, which may omit some runs, the so-called *max-rank* runs, that are essential for our other optimizations.[1]

The main idea of MaxRank is that a set of runs of $\mathcal{B} = \text{Schewe}(\mathcal{A})$ (including super-tight runs) that assign different ranks to non-trunk states is represented by a single, "maximal," not necessarily super-tight (but having the same rank), run in $\mathcal{C} = \text{MaxRank}(\mathcal{A})$. We call such runs *max-rank runs*. More concretely, when moving from the waiting to the tight part, $\mathcal{C}$ needs to correctly guess a rank that is needed on an accepting run and the first tight core of a trunk of the run. The ranks of the rest of states are made maximal. Then, the tight part of $\mathcal{C}$ contains for each macrostate and symbol at most two successors: one via η_3 and one via η_4. Loosely speaking, the η_3-successor keeps all ranks as high as possible, while the η_4-successor decreases the rankings of all non-accepting states in O (and can therefore help emptying O, which is necessary for an accepting run).

Before we give the construction, let us first provide some needed notation. We further use $(S, O, f, i) \leq (S, O, g, i)$ to denote that $f \leq g$ and similarly for $<$ (note that non-ranking components of the macrostates need to match). In the definitions, given a set of macrostates R from Q_2, we use $\max_f\{(S, O, f, i) \in R\}$ to denote the set of maximal elements of the partial order $\leq$ on macrostates in R.

The construction is then formally defined as $\text{MaxRank}(\mathcal{A}) = (Q_1 \cup Q_2, \Sigma, \eta, I', F')$ with $\eta = \delta_1 \cup \eta_2 \cup \eta_3 \cup \eta_4$ such that $Q_1, Q_2, I', F', \delta_1$ are the same as in Schewe. Let $\mathcal{B} = \text{Delay}(\mathcal{A}) = (\cdot, \Sigma, \delta_1 \cup \theta_2 \cup \delta_3, \cdot, \cdot)$ where δ_1, θ_2, and δ_3 are defined as in Delay. We define an auxiliary transition function that keeps macrostates satisfying conditions RankSim′ and SuccRank as follows:

$$\Delta^\bullet(q, a) = \{q' \mid q' \in \theta_2(q, a) \wedge \text{RankSim}'(q') \wedge \text{SuccRank}(q'))\}. \tag{12.12}$$

(We note that q is from the waiting and q' is from the tight part of $\mathcal{B}$.) Given a macrostate (S, O, f, i) and a symbol $a \in \Sigma$, we define the maximal successor ranking, denoted by $f'_{max} = \textit{max-rank}((S, O, f, i), a)$, as follows. Consider $q' \in \delta(S, a)$ and the rank $r = \min\{f(s) \mid s \in \delta^{-1}(q', a) \cap S\}$. Then

- $f'_{max}(q') := r - 1$ if r is odd and $q' \in F$ and
- $f'_{max}(q') := r$ otherwise.

Let δ_3 be the transition function of the tight part of Schewe$(\mathcal{A})$. We can now proceed to the definition of the missing components of MaxRank$(\mathcal{A})$:

- $\eta_2(S, a) := \max_{f'}\{(S', \emptyset, f', 0) \in \Delta^\bullet(S, a)\}$.
- $\eta_3((S, O, f, i), a)$: Let $f'_{max} = \textit{max-rank}((S, O, f, i), a)$. Then, we set
  - $\eta_3((S, O, f, i), a) := \{(S', O', f'_{max}, i')\}$ when $(S', O', f'_{max}, i') \in \delta_3((S, O, f, i), a)$ (i.e., if f'_{max} is tight) and

[1] We believe that this property was not originally intended by the author, since it is not addressed in the proof. As far as we can tell, the construction is correct, although the original argument of the proof in [250] needs to be corrected.

– $\eta_3((S, O, f, i), a) := \emptyset$ otherwise.

- $\eta_4((S, O, f, i), a)$: Let $\eta_3((S, O, f, i), a) = \{(S', P', h', i')\}$ and let
  – $f' = h' \lhd \{u \mapsto h'(u) - 1 \mid u \in P' \setminus F\}$ and
  – $O' = P' \cap f'^{-1}(i')$.

  Then, if $i' \neq 0$, we set $\eta_4((S, O, f, i), a) := \{(S', O', f', i')\}$. Otherwise, we set $\eta_4((S, O, f, i), a) := \emptyset$.

Note that η_3 and η_4 are deterministic (though not complete), so we will sometimes use the notation $(S', O', f', i') = \eta_3((S, O, f, i), a)$.

MaxRank differs from ScheweRedAvgOut in the definition of η_2 and η_4. In particular, in the η_4 of ScheweRedAvgOut (named γ_4 therein), the condition that only non-accepting states ($u \in P' \setminus F$) decrease rank is omitted. Instead, the rank of all states in P' is decreased by one, which might create a "ranking" that is actually not a ranking according to the definition (since an accepting state is given an odd rank), so the target macrostate is omitted from the complement. Due to this, some max-rank runs may also be removed. Our construction preserves max-rank runs, which makes the proof of the theorem significantly more involved.

Theorem 12.2.1. *Let $\mathcal{A}$ be a BA and $\mathcal{C} = \textsc{MaxRank}(\mathcal{A})$. Then $\mathcal{L}(\mathcal{C}) = \overline{\mathcal{L}(\mathcal{A})}$.*

Proof. W.l.o.g. assume that $\mathcal{A}$ is complete. Let $\mathcal{B} = \textsc{Schewe}(\mathcal{A})$. Showing $\mathcal{L}(\mathcal{C}) \subseteq \mathcal{L}(\mathcal{B})$ is easy (the transitions of $\mathcal{C}$ are contained in the transitions of $\mathcal{B}$). Next, we show that $\mathcal{L}(\mathcal{C}) \supseteq \mathcal{L}(\mathcal{B})$.

Let $\alpha \in \mathcal{L}(\mathcal{B})$ and ρ be a super-tight run (from Lemma 12.1.1, we know that a super-tight run exists) of $\mathcal{B}$ over $\alpha = \alpha_0\alpha_1\alpha_2 \ldots$ s.t.

$$\rho = S_0 \ldots S_m(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})(S_{m+2}, O_{m+2}, f_{m+2}, i_{m+2}) \ldots$$

Let $\tau = C_{m+1}C_{m+2} \ldots$ be a trunk of ρ. We will construct the run

$$\rho' = S_0 \ldots S_m(S_{m+1}, O'_{m+1}, f'_{m+1}, i'_{m+1})(S_{m+2}, O'_{m+2}, f'_{m+2}, i'_{m+2}) \ldots$$

of $\mathcal{C}$ on α in the following way (note that the S-components of the macrostates traversed by ρ and ρ' are the same):

(i) For the transition from the waiting to the tight part, we set $O'_{m+1} := \emptyset$ and $i'_{m+1} := 0$. The ranking f'_{m+1} is set as follows. Let r be the rank of ρ (remember that ρ is super-tight). We first construct an auxiliary (tight) ranking

$$g = f_{m+1} \lhd \{u \mapsto \max\{r - 1, f_{m+1}(u)\} \mid u \in S_{m+1} \setminus C_{m+1}\}. \tag{12.13}$$

Note that C_{m+1} is also a tight core of g. There are now two possible cases:

(a) $(S_{m+1}, \emptyset, g, 0) \in \eta_2(S_m, \alpha_m)$: If this holds, we set $f'_{m+1} := g$.

(b) Otherwise, $\eta_2(S_m, \alpha_m)$ contains at least one macrostate with a ranking h s.t. $g < h$. We pick an arbitrary such ranking h from $\eta_2(S_m, \alpha_m)$ and set $f'_{m+1} := h$. Note that C_{m+1} is also a tight core of h.

Note that $\eta_2(S_m, \alpha_m)$ contains at least one macrostate $(S_{m+1}, \emptyset, h, 0)$ with the rank r such that $g \leq h$. This follows from the fact that the reductions SuccRank and RankSim only remove macrostates that do not occur on super-tight runs and that $(S_{m+1}, \emptyset, g, 0)$ is not removed using the reductions. The latter follows from (12.5) and (12.9).

(ii) Let $k > m$ and i_* be such that $(\cdot, \cdot, f_*, i_*) = \eta_4((S_k, O'_k, f'_k, i'_k), \alpha_k)$. Then,

- we set $(S_{k+1}, O'_{k+1}, f'_{k+1}, i'_{k+1}) := \eta_4((S_k, O'_k, f'_k, i'_k), \alpha_k)$ if $O_{k+1} = \emptyset$, $i_* = i_{k+1}$, and $f_* \geq f_{k+1}$,
- otherwise, we set $(S_{k+1}, O'_{k+1}, f'_{k+1}, i'_{k+1}) := \eta_3((S_k, O'_k, f'_k, i'_k), \alpha_k)$.

Intuitively, ρ' simulates the super-tight run ρ of $\mathcal{B}$ with the difference that (i) the transition from the waiting to the tight part sets the ranks of all non-core states to $r - 1$, (ii) in the tight part, ρ' keeps taking the maximizing η_3 transitions until it happens that ρ' is stuck with emptying some O, in which case, the ranks of all non-accepting states in O are decreased (the η_4 transition).

First, we prove that the run ρ' constructed according to the procedure above is infinite. Intuitively, there are two possibilities how the construction of ρ' can break: (i) the macrostate $(S_{m+1}, \emptyset, f'_{m+1}, 0)$ is not in $\eta_2(S_m, \alpha_m)$, (ii) $\eta_3((S_m, O_m, f_m, i_m), \alpha_m) = \emptyset$, or (iii) $\eta_4((S_m, O_m, f_m, i_m), \alpha_m) = \emptyset$.

Claim 7: *For every $k > m$ the following conditions hold:*

(i) the macrostate $\rho'(k)$ is well defined,

(ii) $f'_k \geq f_k$, and

(iii) C_k is a tight core of $\rho'(k)$.

Proof By induction on the position $k > m$ in ρ'.

$k = m + 1$:

(i) Proving $(S_{m+1}, \emptyset, f'_{m+1}, 0) \in \eta_2(S_m, \alpha_m)$: This easily follows from the construction of $(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})$ given above.

(ii) Proving $f'_{m+1} \geq f_{m+1}$: This, again, easily follows from the construction of the macrostate $(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})$. In particular, the g constructed in (12.13) satisfied the property $g \geq f_{m+1}$ and the f'_{m+1} constructed from it satisfies $f'_{m+1} \geq g$.

(iii) We have that C_{m+1} is a tight core of $(S_{m+1}, O_{m+1}, f_{m+1}, i_{m+1})$. From the definition of f'_{m+1} we directly obtain that C_{m+1} is also a tight core of $(S_{m+1}, O'_{m+1}, f'_{m+1}, i'_{m+1})$.

$k + 1$: Suppose the claim holds for k.

(i) (and (iii)) For proving $\rho'(k + 1)$ is well-defined, from the construction, we need to prove the following:

- *If $\rho'(k+1)$ is the η_3-successor of $\rho'(k)$, we need to show that $\eta_3((S_k, O'_k, f'_k, i'_k), \alpha_k) \neq \emptyset$.*

  This condition can be proved by showing that the ranking given as $f'_{max} = \mathit{max\text{-}rank}((S_k, O'_k, f'_k, i'_k), \alpha_k)$ is tight. From the induction hypotheses ("$f'_k \geq f_k$"

and "C_k is a tight core of f'_k"), we know that f_k and f'_k coincide on states from C_k. Further, from Lemma 12.1.2, it holds that for every state $q_k \in C_k$ there is a state $q_{k+1} \in C_{k+1}$ such that $f_k(q_k) = f_{k+1}(q_{k+1})$. From the construction of f'_{max}, we can conclude that it also holds that $f'_k(q_k) = f'_{max}(q_{k+1})$ (which proves (iii)). Using induction hypothesis one more time ("f'_k is tight"), we can conclude that f'_{max} is also tight.

- *If $O_{k+1} = \emptyset$, $i_{k+1} = i'_{k+1}$, and $f'_{k+1} \geq f_{k+1}$ hold at the same time (i.e., $\rho'(k+1)$ is the η_4-successor of $\rho'(k)$), we need to show that $\eta_4((S_k, O'_k, f'_k, i'_k), \alpha_k) \neq \emptyset$.*

  Above, we have already shown that $\eta_3((S_k, O'_k, f'_k, i'_k), \alpha_k) \neq \emptyset$. From the definition, in order for $\eta_4((S_k, O'_k, f'_k, i'_k), \alpha_k) = \emptyset$, it would need to hold that $i'_{k+1} = 0$. Since our assumption is that $\mathcal{A}$ is complete and we know that ρ is accepting, it needs to hold that at every position $j > m$, we have $f_j(q) > 0$ for any state $q \in S_j$ (otherwise, if q appeared in the O-component of some macrostate in ρ, it would never disappear and so ρ could not be accepting).

  The proof of (iii) easily follows from the previous step for η_3, since the ranking function of the result of η_4 differs from the one for η_3 only on states from the O-component, which are even and therefore, by definition, not in a tight core.

(ii) Proving $f'_{k+1} \geq f_{k+1}$ assuming the induction hypothesis $f'_k \geq f_k$:

- *If $\rho'(k+1)$ is the η_3-successor of $\rho'(k)$, $f'_{k+1} \geq f_{k+1}$* follows immediately from the fact that the η_3 transition function yields the maximal successor ranking.
- *If $O_{k+1} = \emptyset$, $i_{k+1} = i'_{k+1}$, and $f'_{k+1} \geq f_{k+1}$ hold at the same time (i.e., $\rho'(k+1)$ is the η_4-successor of $\rho'(k)$), $f'_{k+1} \geq f_{k+1}$* is already a condition for η_4 to be taken. ■

Next, we will show that ρ' is accepting, i.e., that O-component of macrostates in ρ' is emptied infinitely many times. For the sake of contradiction, assume that ρ' is not accepting, i.e., for some $\ell > m$, it happens that for all $k \geq \ell$ it holds that $O'_k \neq \emptyset$ and $i'_k = i'_\ell$ (the run is "stuck" at some i' and cannot empty O'). We will show that if ρ' is "stuck" at some i', it will contain infinitely many macrostates obtained using an η_4 transition. An η_4 transition decreases ranks of all non-final states in O' and, as a consequence, removes such tracked runs from O'. Therefore, if the rank of some run in O' is infinitely often not decreased by η_4, there needs to be a corresponding run of $\mathcal{A}$ with infinitely many occurrences of an accepting state, so it would need to hold that $\alpha \in \mathcal{A}$, leading to a contradiction.

Let us now prove the previous reasoning more formally. First, we show that ρ' needs to contain infinitely many occurrences of η_4-obtained macrostates. Since ρ is accepting, it satisfies infinitely often the condition that O is empty and $i = i'_\ell$. To satisfy the condition for executing an η_4 transition, we need to show that for infinitely many k it in addition holds that $f'_k = f'_{max} \lhd \{q \mapsto f'_{max}(q) - 1 \mid u \in O'_k \setminus F\} \geq f_k$ where f'_{max} is as in the definition of η_3. In particular, we will show that for infinitely many k, we will have $i_k = i'_\ell$, $O_k = \emptyset$, and $\forall q \in O'_k \setminus F : f'_{max}(q) > f(q)$ (from Claim 7 we already know that $f'_{max} \geq f_k$).

Let $p > \ell$ be a position such that $i_{p-1} \neq i'_\ell$, $O_{p-1} = \emptyset$, and $i_p = i'_\ell$, i.e., a position at which run ρ started emptying runs with rank i'_ℓ. Because ρ is accepting, there is a position $k \geq p$ such that $\rho(k) = (S_k, \emptyset, f_k, i'_\ell)$, therefore, the ranks of all runs tracked in O_p were decreased (otherwise, O_k could not be empty). Consider the following claim.

Claim 8: $\forall q \in O'_k : f_k(q) < f'_k(q)$

Proof The weaker property $f_k \leq f'_k$ follows from Claim 7. We prove the strict inequality for states in O'_k by contradiction. Assume that $f_k(q) = f'_k(q)$ for some $q \in O'_k$. Then there needs to be a predecessor s of q in S_p such that $f_p(s) = i'_\ell$ and so $s \in O_p$. But since $q \notin O_k$, then somewhere between p and k, the rank of the run in $\mathcal{A}$ must have been decreased. Therefore, $f_k(q) < f'_k(q)$. ■

From Claim 8 and the fact that $f'_k(q) \leq f'_{max}$ it follows that $\forall q \in O'_k \setminus F : f'_{max}(q) > f(q)$, so an η_4 transition was taken infinitely often in ρ'.

The last thing to show is that when η_4 is taken infinitely often, O' will be eventually empty. The condition does not hold only in the case when the rank of a run of $\mathcal{A}$ tracked in O' is infinitely often not decreased because it is represented in O' by a final state $q \in O' \cap F$. But then $\mathcal{A}$ contains a run over α that touches an accepting state infinitely often, so $\alpha \in \mathcal{L}(\mathcal{A})$, which is a contradiction. □

Note that MaxRank is incompatible with RankRestr since RankRestr optimizes the transitions in the tight part of the complement BA, and these transitions are abstracted in MaxRank.

12.2.6 Backing Off

Our final optimization, called BackOff, is a strategy for guessing when our optimized rank-based construction is likely (despite the optimizations) to generate too many states and when it might be helpful to give up and use a different complementation procedure instead. We evaluate this after the initial phase of Schewe, constructing δ_2 (η_2 in MaxRank, θ_2 in Delay; we will just use δ_2 now) finishes. In particular, we provide a set of pairs $\{(\mathit{StateSize}_i, \mathit{RankMax}_i)\}_{i \in \mathcal{I}}$ for an index set $\mathcal{I}$. We then check (after δ_2 is constructed) that for all $(S, O, f, i) \in \mathrm{img}(\delta_2)$ and all $i \in \mathcal{I}$ it holds that either $|S| < \mathit{StateSize}_i$ or $\mathit{rank}(f) < \mathit{RankMax}_i$. If for some (S, O, f, i) and i the condition does not hold, we terminate the construction and execute a different, *surrogate*, procedure.

12.3 Experimental Evaluation

Used tools and evaluation environment. We implemented the optimizations described in the previous sections in a tool called Ranker[2] in C++ (we tested the correctness of our implementation using Spot's `autcross` on all BAs in our benchmark; in many cases it did not finish, but for those it finished, it never reported an error). We compared our complementation approach with other state-of-the-art tools, namely, GOAL [277] (including the Fribourg plugin [16]), Spot 2.9.3 [101], ROLL [186], Seminator 2 [46] and LTL2dstar 0.5.4 [175]. All tools were set to the mode where they output an automaton with the standard state-based Büchi acceptance condition. We note that some of the tools are aimed at complementing more general flavors of ω-automata, such as Seminator 2 focusing on generalized transition-based Büchi automata. 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. The timeout was set to 5 minutes.

Dataset. As the source of our main benchmark, we took the 11,000 BAs used in [276], which were randomly generated using the Tabakov-Vardi approach [265] over a two letter

[2] Ranker is available at `https://github.com/vhavlena/ba-inclusion`

alphabet, starting with 15 states, with various different parameters (see [276] for more details). In preprocessing, the automata were reduced using a combination of Rabit [206] and Spot's `autfilt` (using the `-high` simplification level) and converted to the HOA format [23]. From this set, we removed automata that are (i) semi-deterministic, (ii) inherently weak, or (iii) unambiguous, since for these kinds of automata there exist more efficient complementation procedures than for unrestricted BAs [47, 46, 50, 188]. Moreover, we removed BAs with an empty language or empty language of complement (since the complement has 1 state only). We were left with **2,393** *hard* automata. (In Section 12.3.3, we also present additional results on a less challenging benchmark used in [46], containing BAs obtained by translation from LTL formulae.)

Selection of optimizations. We use two settings of Ranker with different optimizations turned on. Since the RankRestr and MaxRank optimizations are incompatible, the main difference between the settings is which one of those two they use. The particular optimizations used in the settings are the following:

$$\textsc{Ranker}_{\textsc{MaxR}} = \textsc{Delay} + \textsc{SuccRank} + \textsc{RankSim}' + \textsc{MaxRank}$$
$$\textsc{Ranker}_{\textsc{RRestr}} = \textsc{Delay} + \textsc{SuccRank} + \textsc{RankSim}' + \textsc{RankRestr} + \textsc{Prg}_{di}$$

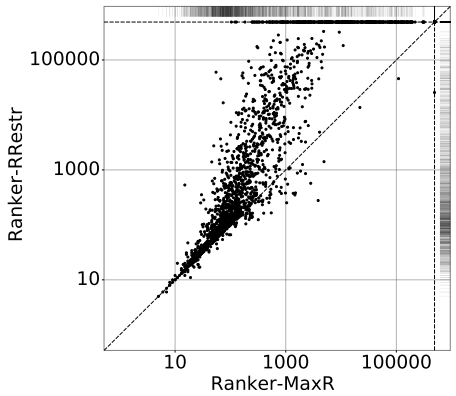
(The $\textsc{Prg}_{di}$ optimization is taken from Chapter 11.) Note that the two settings include all respective compatible optimizations. Due to space constraints, we cannot give a detailed analysis of the effect of individual optimizations on the size of the obtained complement automaton. Let us, at least, give a bird's-eye view. The biggest effect has MaxRank, followed by Delay—their use is key to obtaining a small state space. The rest of the optimizations are less effective, but they still remove a significant number of states.

12.3.1 Comparison of Rank-Based Procedures

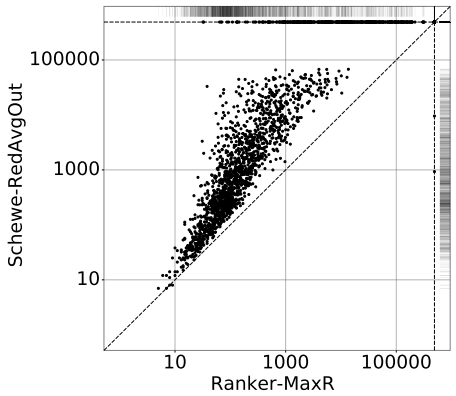
First, we evaluated how our optimizations reduce the generated state space, i.e., we compared the sizes of generated complements without any postprocessing. Such a use case represents applications like testing inclusion, equivalence, or universality of BAs, where postprocessing the output is irrelevant.

More precisely, we first compared the sizes of automata produced by our settings $\textsc{Ranker}_{\textsc{MaxR}}$ and $\textsc{Ranker}_{\textsc{RRestr}}$ to see which of them behaves better (cf. Figure 12.4a) and then we compared $\textsc{Ranker}_{\textsc{MaxR}}$, which had better results, with the $\textsc{Schewe}_{\textsc{RedAvgOut}}$ procedure implemented in GOAL (parameters `-m rank -tr -ro`). Scatter plots of the results are given in Figure 12.4b and summarizing statistics in the upper part of Table 12.2.

We note that although $\textsc{Ranker}_{\textsc{MaxR}}$ produces in the vast majority of cases (1,810) smaller automata than $\textsc{Ranker}_{\textsc{RRestr}}$, there are still a few cases (109) where $\textsc{Ranker}_{\textsc{RRestr}}$ outputs a smaller result (in 1 case this is due to the timeout of $\textsc{Ranker}_{\textsc{MaxR}}$). The comparison with $\textsc{Schewe}_{\textsc{RedAvgOut}}$ shows that our optimizations indeed have a profound effect on the size of the generated state space. Note that although the mean and maximum size of complements produced by $\textsc{Ranker}_{\textsc{MaxR}}$ and $\textsc{Ranker}_{\textsc{RRestr}}$ are larger than those of $\textsc{Schewe}_{\textsc{RedAvgOut}}$, this is because for cases where the complement would be large, the run of $\textsc{Schewe}_{\textsc{RedAvgOut}}$ in GOAL timeouted before it could produce a result. Therefore, the median is a more meaningful indicator, and it is significantly (three to four times) lower for both $\textsc{Ranker}_{\textsc{MaxR}}$ and $\textsc{Ranker}_{\textsc{RRestr}}$.



(a) RANKER$_{\text{MAXR}}$ vs RANKER$_{\text{RRESTR}}$



(b) RANKER$_{\text{MAXR}}$ vs SCHEWE$_{\text{REDAVGOUT}}$

Figure 12.4: Evaluation of the effectiveness of our optimizations on the generated state space (axes are logarithmic). The horizontal and vertical dashed lines represent timeouts.

Table 12.1: Run times of the tools in seconds

method	mean	med.	std. dev
RANKER$_{\text{MAXR}}$	10.21	0.84	28.43
RANKER$_{\text{MAXR}}$+BO	9.40	3.03	16.00
PITERMAN	7.47	6.03	8.46
SAFRA	15.49	7.03	35.59
SPOT	1.07	0.02	8.94
FRIBOURG	19.43	10.01	32.76
LTL2DSTAR	4.17	0.06	22.19
SEMINATOR 2	11.41	0.37	34.97
ROLL	42.63	14.92	67.31

12.3.2 Comparison with Other Approaches

Further, we evaluated the complements produced by RANKER$_{\text{MAXR}}$ and other approaches. In this setting, we focused the evaluation on the size of the output BA *after* postprocessing (we, again, used `autfilt` with simplification level `-high`; we denote this using "+PP"). We evaluated the following algorithms: SAFRA [248] implemented in GOAL (parameter `-m safra`), its optimization PITERMAN [228] implemented in GOAL (parameter `-m piterman`) and the optimization implemented in LTL2DSTAR [175], FRIBOURG [16] implemented as a plugin of GOAL, the determinization-based complementation implemented in SPOT (optimized construction of Redziejowski [241]), a learning-based algorithm [187] implemented in ROLL, and a semideterminization-based algorithm [47] in SEMINATOR 2.

In Figure 12.5, we give scatter plots of selected comparisons. In particular, due to space constraints, we omitted the results for SAFRA, SPOT, and LTL2DSTAR, which on average performed slightly worse than PITERMAN. We give summarizing statistics in the lower part of Table 12.2 and the run times in Table 12.1.

Let us now discuss the data in the lower part of the table. In the left-hand side, we can see that the mean and median size of BAs obtained by RANKER$_{\text{MAXR}}$ are both the lowest with the exception of ROLL. ROLL implements a learning-based approach, which means that it works on the level of the *language* of the input BA instead of the *structure*. Therefore, it can often find a much smaller automaton than other approaches. Its practical

Table 12.2: Statistics for our experiments. The upper part compares different optimizations of the rank-based procedure (no postprocessing). The lower part compares our approach with other methods (with postprocessing). "BO" denotes the BACKOFF optimization. In the left-hand side of the table, the column "**med.**" contains the median, "**std. dev**" contains the standard deviation, and "**TO**" contains the number of timeouts (5 mins). In the right-hand side of the table, we provide the number of cases where our tool (RANKER$_{\text{MAXR}}$ without postprocessing in the upper part and with postprocessing in the lower part) was strictly better ("**wins**") or worse ("**loses**"). The "**(TO)**" column gives the number of times this was because of the timeout of the loser.

method	max	mean	med.	std. dev	TO	wins	(TO)	loses	(TO)
RANKER$_{\text{MAXR}}$	319 119	8 051.58	185	28 891.4	360	—	—	—	—
RANKER$_{\text{RRESTR}}$	330 608	9 652.67	222	32 072.6	854	1810	(495)	109	(1)
SCHEWE$_{\text{REDAVGOUT}}$	67 780	5 227.3	723	10 493.8	844	2030	(486)	3	(2)
RANKER$_{\text{MAXR}}$	1 239	61.83	32	103.18	360	—	—	—	—
RANKER$_{\text{MAXR}}$+BO	1 706	73.65	33	126.8	17	—	—	—	—
PITERMAN	1 322	88.30	40	142.19	12	1 069	(3)	469	(351)
SAFRA	1 648	99.22	42	170.18	158	1 171	(117)	440	(319)
SPOT	2 028	91.95	38	158.13	13	907	(6)	585	(353)
FRIBOURG	2 779	113.03	36	221.91	78	996	(51)	472	(333)
LTL2DSTAR	1 850	88.76	41	144.09	128	1 156	(99)	475	(331)
SEMINATOR 2	1 772	98.63	33	191.56	345	1 081	(226)	428	(241)
ROLL	1 313	21.50	11	57.67	1 106	1 781	(1 041)	522	(295)

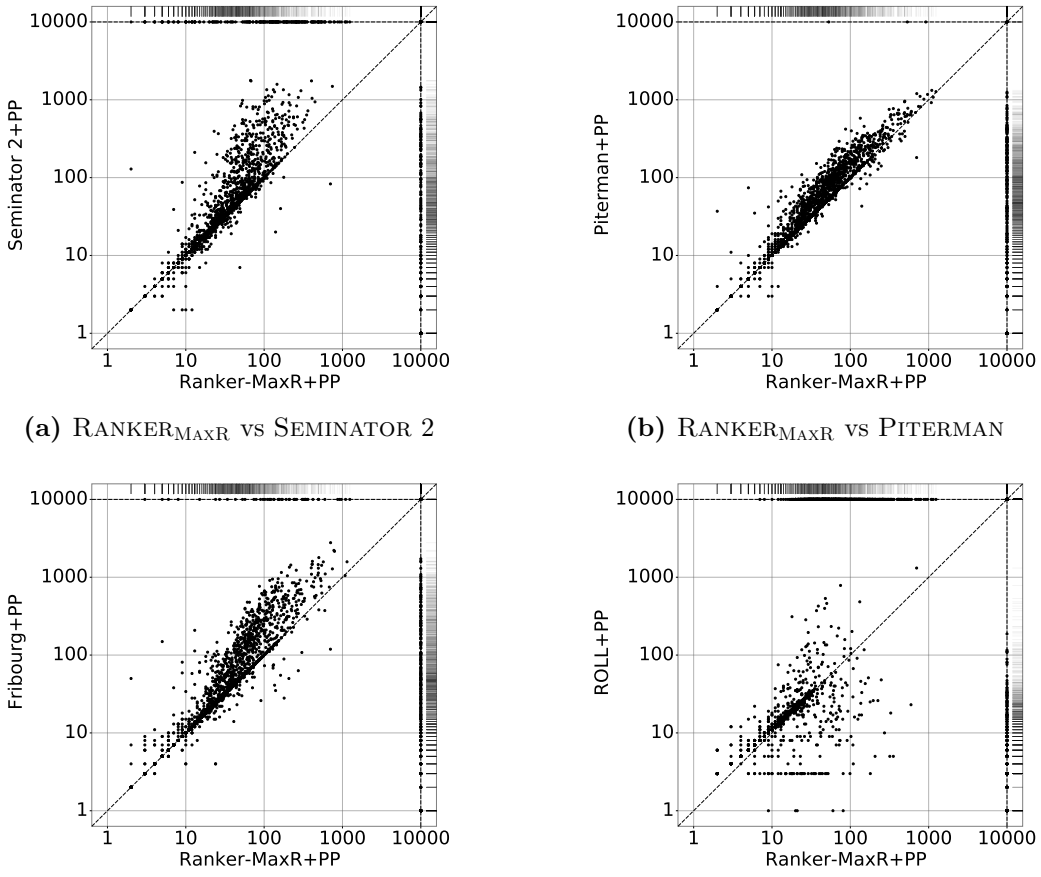
time complexity, however, seems to grow much faster with the number of states of the output BA than other approaches (ROLL had by far the largest mean and median run time, as shown in Table 12.1). RANKER$_{\text{MAXR}}$ by itself had more timeouts than other approaches, but when used with the BACKOFF strategy, is on par with PITERMAN and SPOT.

In the right-hand side of the table, we give the numbers of times where RANKER$_{\text{MAXR}}$ gave strictly smaller and strictly larger outputs, respectively. Here, we can see that the output of RANKER$_{\text{MAXR}}$ is often at least as small as the output of the other method (this is not in the table, but can be computed as $2,393 - \textbf{loses}$; the loses were caused mostly by timeouts; results with the BACKOFF strategy would increase the number even more) and often a strictly smaller one (the **wins** column). When comparing RANKER$_{\text{MAXR}}$ with *the best result of any other tool*, it obtained a *strictly smaller* BA in 539 cases (22.5 %) and a BA *at least as small* as the best result of any other tool in 1,518 cases (63.4 %). Lastly, we note that there were four BAs in the benchmark that *no tool* could complement and one BA that *only* RANKER$_{\text{MAXR}}$ was able to complement (namely, `new-s-15-r-1.00-f-0.30-72-of-100.ba-red.hoa` with a 919-state-large complement). There was no such a case for any other tool.

Let us now focus on the run times of the tools given in Table 12.1. GOAL-based approaches and ROLL are implemented in Java, which adds a significant overhead to the run time (e.g., the fastest run time of GOAL was 3.15 s; it is hard to predict how their performance would change if they were reimplemented in a faster language), while the other approaches are implemented in C++.

BackOff. Our BACKOFF setting in the experiments used the set of constraints

$$\{(\mathit{StateSize}_1 = 9, \mathit{RankMax}_1 = 5), (\mathit{StateSize}_2 = 8, \mathit{RankMax}_2 = 6)\}$$



(a) RANKER$_{\text{MAXR}}$ vs SEMINATOR 2 **(b)** RANKER$_{\text{MAXR}}$ vs PITERMAN

(c) RANKER$_{\text{MAXR}}$ vs FRIBOURG **(d)** RANKER$_{\text{MAXR}}$ vs ROLL

Figure 12.5: Comparison of the sizes of the BAs constructed using our optimized rank-based construction and other approaches. Timeouts are on the dashed lines.

and PITERMAN as the surrogate algorithm. The BACKOFF strategy was executed 873 times and managed to decrease the number of timeouts of RANKER$_{\text{MAXR}}$ from 360 to 17 (row RANKER$_{\text{MAXR}}$+BO in Table 12.2).

Discussion. The results of our experiments show that our optimizations are key to making rank-based complementation competitive to other approaches in practice. Furthermore, with the optimizations, the obtained procedure in the majority of cases produces a BA at least as small as a BA produced by any other approach, and in a large number of cases *the smallest* BA produced by any existing approach. We emphasize the usefulness of the BACKOFF heuristic: as there is no clear "best" complementation algorithm—different techniques having different strengths and weaknesses—knowing which technique to use for an input automaton is important in practice. In Table 12.3, we give a modification of the right-hand side of Table 12.2 giving wins and loses for RANKER$_{\text{MAXR}}$+BO. It seems that the combination of these two completely different algorithms yields a quite strong competitor.

Table 12.3: Wins and loses for RANKER$_{\text{MAXR}}$+BO

method	wins	(TO)	loses	(TO)
PITERMAN	1 160	(4)	112	(9)
SAFRA	1 255	(147)	222	(6)
SPOT	985	(8)	328	(12)
FRIBOURG	1 076	(71)	287	(10)
LTL2DSTAR	1 208	(118)	272	(7)
SEMINATOR 2	1 236	(333)	253	(5)
ROLL	1 923	(1 096)	360	(7)

Table 12.4: Statistics for our experiments on the LTL benchmark (see the explanation of the columns in the description of Table 12.2).

method	max	mean	med.	std. dev	TO	wins	(TO)	loses	(TO)
RANKER$_{\text{MAXR}}$	43 527	357.71	29	2 510.31	5	—	—	—	—
RANKER$_{\text{RRESTR}}$	214 946	1 948.26	33	13 928	17	281	(12)	35	(0)
SCHEWE$_{\text{REDAVGOUT}}$	33 345	665.27	35	3 081.89	9	292	(5)	83	(1)
RANKER$_{\text{MAXR}}$	330	20.07	10	32.86	5	—	—	—	—
RANKER$_{\text{MAXR}}$+BO	440	21.09	11	38.28	2	—	—	—	—
PITERMAN	436	21.98	14.5	30.66	2	287	(1)	76	(4)
SAFRA	361	30.39	17	44.24	14	330	(10)	56	(1)
SPOT	151	14.52	10	17.71	0	138	(0)	195	(5)
FRIBOURG	212	12.85	9	16.22	1	56	(1)	238	(5)
LTL2DSTAR	223	21.05	13	24.17	6	249	(5)	111	(4)
SEMINATOR 2	233	14.62	10	19.97	1	121	(1)	218	(5)
ROLL	96	13.81	11	10.62	3	243	(3)	150	(5)

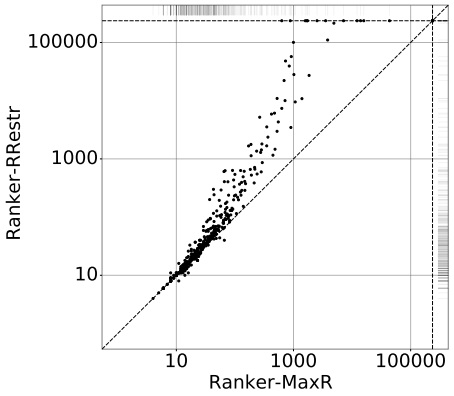
12.3.3 Experimental Results for BAs from LTL formulae

In this part, we give also experimental evaluation of our optimizations on the benchmark from [46]. The benchmark contains automata obtained from LTL formulae (i) from literature (221) and (ii) randomly generated (1500). (We are not aware of the motivation behind complementing BAs obtained from LTL formulae—it is a well known fact negating the formula and constructing a BA directly for the negation only increases the size of the BA *linearly* instead of *exponentially*). From the benchmark, we selected only 414 *hard* BAs (in the same way as in the previous). Note that although we selected only 414 *hard* instances, their structure is still simpler than the structure of the BAs considered at the beginning of Section 12.3, since LTL does not have the full power of ω-regular languages (this difference in the difficulty can be seen from the summary statistics in Table 12.4). The timeout was again set to 5 minutes.

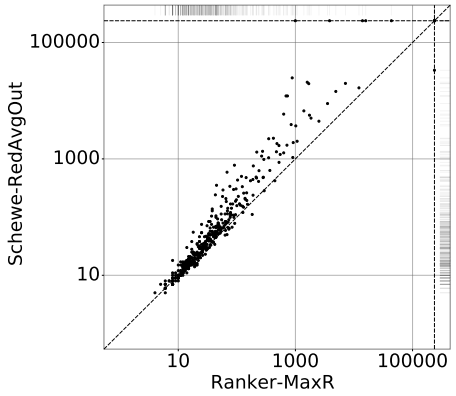
In Table 12.4, we can see that the average sizes of the complements produced by RANKER$_{\text{MAXR}}$, compared to the other tools, are larger than in our main benchmark, the median is, however, comparable. We believe that the larger average size is due to the two following facts: (i) that BAs from LTL formulae have a simpler structure than general BAs that is more suitable for the other approaches and (ii) our implementation does not take advantage of the symbolic alphabets present in the benchmark (we immediately translate the alphabet to an explicit alphabet, neglecting any relations among the symbols). Moreover, note that RANKER$_{\text{MAXR}}$ is no longer a clear winner here. In particular,

Table 12.5: Run times of the tools on the LTL benchmark (in seconds)

method	mean	med.	std. dev
RANKER$_{\text{MAXR}}$	1.99	0.04	16.51
RANKER$_{\text{MAXR}}$+BO	1.27	0.05	8.62
PITERMAN	6.65	5.62	3.73
SAFRA	8.37	5.80	13.45
SPOT	0.06	0.02	0.71
FRIBOURG	7.22	5.48	13.22
LTL2DSTAR	0.11	0.02	0.89
SEMINATOR 2	0.08	0.02	0.83
ROLL	7.28	2.74	16.06



(a) RANKER$_{\text{MAXR}}$ VS RANKER$_{\text{RRESTR}}$

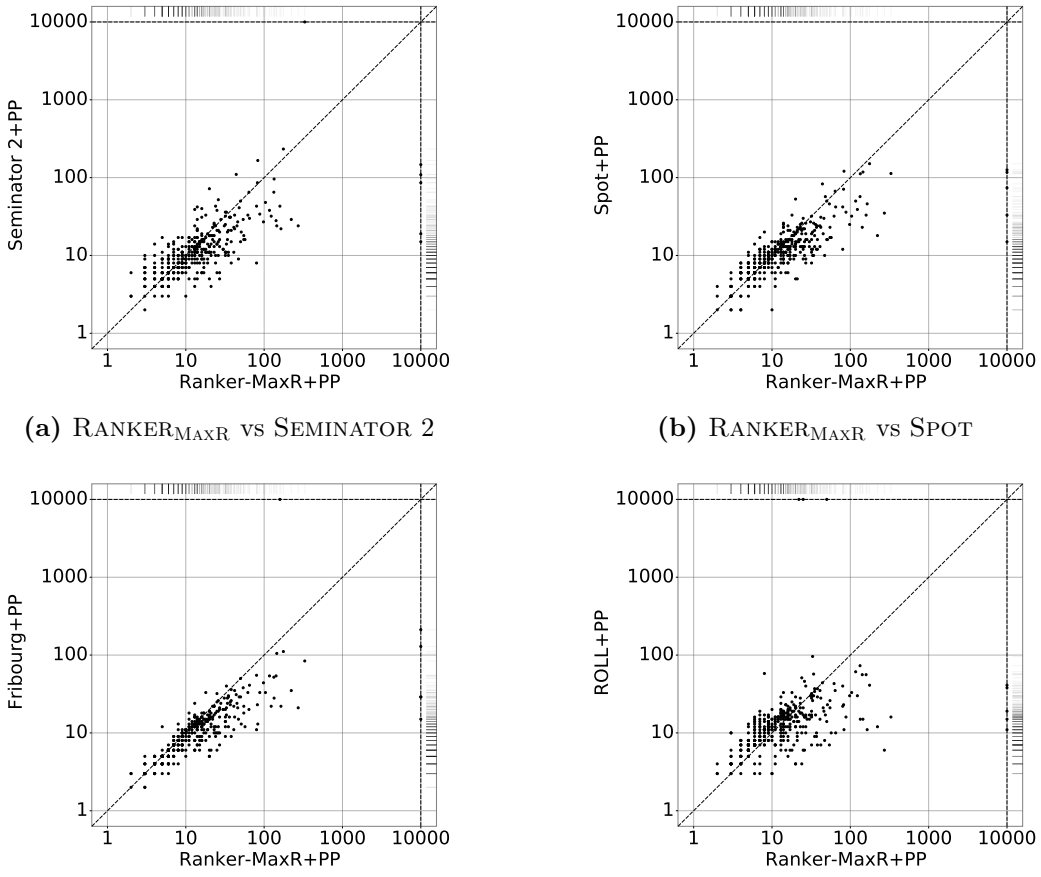


(b) RANKER$_{\text{MAXR}}$ VS SCHEWE$_{\text{REDAVGOUT}}$

Figure 12.6: Evaluation of the effectiveness of our optimizations on the generated state space for the LTL benchmark. Both axes are logarithmic. A point on the horizontal or vertical dashed lines represents a timeout.

it is now comparable to SPOT and SEMINATOR 2 (both provide smaller automata slightly more often); FRIBOURG is the clear winner on this benchmark. It is also interesting to see that the results for PITERMAN are significantly worse when compared to the other tools than in our main experiments in the previous part. One possible explanation might be that the benchmarks from LTL formulae contain symbolic alphabets; as far as we know, the implementation of PITERMAN in GOAL turns such an alphabet into an explicit one, so it cannot exploit the internal structure of symbols on transitions). Scatter plots comparing rank-based approaches are in Figure 12.6. Furthermore, scatter plots comparing RANKER$_{\text{MAXR}}$ with other approaches are in Figure 12.7. Compared with the scatter plots in Figure 12.5, we substituted PITERMAN with SPOT, which had better results than both PITERMAN and SAFRA.

In Table 12.5, we provide the times needed by the tools. We note that SEMINATOR 2 performs much better on this benchmark, the performance of PITERMAN goes down, and also that ROLL does not perform as bad as in our main benchmark (we believe that this is due to the lower difficulty of this benchmark).



(a) RANKER$_{\text{MAXR}}$ vs SEMINATOR 2 **(b)** RANKER$_{\text{MAXR}}$ vs SPOT

(c) RANKER$_{\text{MAXR}}$ vs FRIBOURG **(d)** RANKER$_{\text{MAXR}}$ vs ROLL

Figure 12.7: Comparison of the size of the BAs constructed using our optimized rank-based construction and other approaches on the LTL benchmark. Timeouts (5 mins) are on the edges.

12.4 Conclusion

We developed a series of optimizations reducing the state space generated in the rank-based complementation of Büchi automata. Our experimental evaluation shows that our approach is competitive with other state-of-the-art complementation techniques and often outperforms them.

There are several possible directions for future work. There are still ways to improve our procedure. In particular, we have ideas about refining RANKSIM to obtain an even larger reduction. Furthermore, DELAY can be further refined by a smarter choice of *when* to perform the transition from the waiting to the tight part of the BA. Currently, this is done when a cycle in the waiting part is closed, which does not need to be the best choice—we could utilize information about the number of successors of selected states to choose a better point.

In this chapter, in order to keep the presentation easier to follow, we focused on BAs with one state-based acceptance condition. Our optimizations can, however, be extended to *generalized BAs* in a straightforward way and to *transition-based generalized BAs* (TGBAs) with a modification of SUCCRANK (final states cannot be considered any more). It is

an open question whether the richer structure of TGBAs brings other opportunities for reductions. We also plan to extend our approach to efficient (TG)BA language inclusion checking, where even more reduction (in the style of [14]) of the state space is possible. The contents of this chapter was published as a technical report [139] and was submitted to CONCUR'21 [140].

Chapter 13

Conclusion and Further Directions

The aim of this thesis was the development of efficient techniques for handling finite automata. In particular, we focused on handling of automata in the context of network intrusion detection, automata in decision procedures, and complementation of Büchi automata. The first part of the thesis was devoted to approximate reduction of NFAs in the context of network intrusion detection. We proposed novel approximate reductions of NFAs that are used in order to reduce consumed resources of HW-accelerated RE pattern matchers. First, we introduced approximate reductions with formal guarantees w.r.t. a probabilistic model. The procedure selects less significant states where the reduction is applied. The choice of states is steered by the error expressed as a probability that an incoming packet is misclassified. Since the optimal approximate reduction is **PSPACE**-complete, we proposed greedy algorithms alleviating the impact of the complexity on NFAs used in the context of intrusion detection. Secondly, we introduced lightweight approximate reductions where a model of the traffic is not required and the reduction is steered directly by a multiset of strings (captured packets). Using our approximate reductions, we were able to obtain the throughput of 100 Gbps and even 400 Gbps on REs obtained from existing NIDSes, which shows a practical impact of our work.

The second part of the thesis focused on efficient handling of finite automata in decision procedures. First, we proposed a symbolic decision procedure for WS2S based on automata terms that implicitly represent tree automata. Our approach allows to construct the terms lazily in order to generate as small state space as an possible. On top of that, we extended our basic decision procedure with optimizations, such as subsumption pruning, continuation, or early termination. We outperform Mona on multiple parametric families of formulae. Second, we investigated the impact of formula preprocessing (e.g., antiprenexing) on the size of intermediate automata obtained during the automata-based decision procedure of WSkS in Mona. The preprocessing is implemented in the form of syntactic rules. In order to increase the precision, we parameterize the rules by an estimation of the automata size obtained using machine learning techniques. Our formula preprocessing significantly improves the overall performance of Mona. Third, we employed automata in string constraint solving. We expressed solving of quadratic equations within the regular model checking framework. Our approach reduces redundancies in the Nielsen proof graph. On top of that, we employed symbolic register transducers in order to obtain an efficient procedure. We obtained promising results showing that our approach is orthogonal to techniques implemented in state-of-the-art solvers.

The third part of the thesis deals with optimizations in rank-based Büchi automata complementation. First, we proposed the optimization removing states with incompatible

rankings w.r.t. direct and delayed simulation computed on the original BA. Then, we took a step further and proposed optimizations aiming (not only) at reducing the maximum rank of a macrostate (and hence removing states and transitions that are not necessary for acceptance of a word). Our experiments show that our techniques significantly improve the original Schewe's algorithm and that they are competitive with other state-of-the-art approaches.

13.1 Further Directions

There are many possible directions for future work in the concerned areas. Note that we already discussed some of these areas at the end of the corresponding chapters. Here, we make just a brief summary.

Regarding the approximate reduction of NFAs, one option is to extend the set of considered reduction operations with a general quotienting of a given NFA w.r.t. an equivalence taking into account the probabilistic model and/or the sample of string. This could lead to some kind of model-driven approximate simulation (e.g., a variation of [93] proposed for probabilistic processes). As another option we mention development of techniques for learning probabilistic automata representing the input network traffic. Note that obtaining a suitable probabilistic automaton for some network traffic is important not only in our application but also, e.g., in detection of anomalies in the context of industrial control systems, as shown, e.g., in [203], which the author of this thesis co-authored but which is already not included into this thesis. This direction is, therefore, interesting from multiple angles.

Concerning WSkS, unlike the classical automata-based procedure, which can be seen as a bottom-up procedure (automata for a given formula are constructed inductively from leaves), our decision procedure based on automata terms can be seen as a top-down procedure. Therefore, a natural question of a combination of these two approaches arises. In particular, we could construct certain subformulae using the bottom-up approach and then use the top-down approach. Future work could focus on the question when to switch from the bottom-up to the top-down approach in order to obtain the best of both worlds. Another direction could also be to investigate automata minimization in the context of our automata-terms-based procedure. Regarding the preprocessing of WSkS formulae, there is still a lot of space in obtaining more accurate models estimating the sizes of automata, e.g., using neural networks instead of the currently used linear regression. Another direction consists in automated tuning of parameters for our preprocessing techniques, possibly based on features of an input formula. In the case of string constraint solving, we could, as a future work, consider encoding of a complete procedure, e.g., Makanin's algorithm [197] or the recompression algorithm [158] into the RMC framework.

Finally, we briefly discuss further directions related to our optimizations of Büchi automata complementation. One option is to extend our approach to other types of ω-automata, e.g., transition-based generalized BAs (our preliminary considerations confirm that it is possible). Further, we could consider a refinement of our techniques for the reduction of the maximum rank in order to obtain a procedure matching the upper bound of complementation algorithms for various subclasses of BAs (e.g., semideterministic BAs). Another direction could cover an extension of our techniques to language inclusion checking. We could adjust the on-the-fly algorithm for language inclusion checking with pruning techniques based on relations taking into account the structure of macrostates. Recall that the on-the-fly language inclusion constructs a product of the first BA with a complement of

the second one while testing emptiness on-the-fly. One could also use simulation between input BAs (direct, delay, fair) to remove macrostates with empty languages (without the construction of reachable parts of the product automaton).

13.2 Publications Related to This Thesis

The results related to this thesis were published in the following papers. The approximate reduction of NFAs with formal guarantees was published in the proceedings of TACAS'18 (Core A) [283] and later the extended version of this paper appeared in the STTT journal [307]. The lightweight approximate reduction appeared in the proceedings of FCCM'19 (Core A) [306]. The decision procedure for WS2S was published in the proceedings of CADE-27 (Core A) [138] and the extended version of this paper was accepted to appear in the Journal of Automated Reasoning [137]. The preprocessing techniques for WSkS were presented in the proceedings of LPAR'20 (Core A) [141]. Our approach to string constraint solving appeared in the proceedings of APLAS'20 (Core B) [76]. Simulations in rank-based Büchi automata complementation were published in the proceedings of APLAS'19 (Core B) [75] and further optimizations leading to an efficient rank-based complementation appeared as a technical report [139] and it was submitted to CONCUR'21 [140]. The author's contribution to the publications is summarized below.

- TACAS'18 [283], STTT journal [307]: development of the main ideas related to approximate reduction and formal guarantees (including formulation of theorems and proofs), implementation, experimental evaluation, a part of writing.
- FCCM'19 [306]: proposal of the topic, development of ideas related to approximate reduction of NFAs, evaluation of the reduction, a part of implementation and writing.
- CADE-27 [138], JAR [137]: development of the main ideas related to automata terms for WS2S (including formulation of theorems and proofs), implementation, experimental evaluation, a part of writing.
- APLAS'19 [75]: development of the main ideas of use of simulations in rank-based complementation (including formulation of theorems and proofs), implementation, experimental evaluation, a part of writing.
- LPAR'20 [141]: development of the main ideas related to static modifications of WSkS formulae, implementation, experimental evaluation, a part of writing.
- APLAS'20 [76]: development of ideas related to string solving using RMC (including formulation of theorems and proofs), implementation, experimental evaluation, a significant part of writing.
- Technical report [139], CONCUR'21 submission [140]: development of the main ideas concerning ranking restrictions (including formulation of theorems and proofs), implementation, a significant part of writing.

Bibliography

[1] IEEE standard for ethernet – amendment 10: Media access control parameters, physical layers, and management parameters for 200 Gb/s and 400 Gb/s operation. IEEE std 802.3bs-2017 (2017)

[2] Cisco annual internet report (2018–2023) white paper. Report, Cisco Systems (2020), https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf

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

[4] Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing simulations over tree automata. In: Proceedings of TACAS'08. LNCS, vol. 4963, pp. 93–108. Springer (2008)

[5] Abdulla, P.A., Holík, L., Kaati, L., Vojnar, T.: A uniform (bi-)simulation-based framework for reducing tree automata. Electronic Notes in Theoretical Computer Science **251**, 27–48 (2009)

[6] Abdulla, P.A.: Regular model checking. International Journal on Software Tools for Technology Transfer **14**(2), 109–118 (2012)

[7] Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of PLDI'17. pp. 602–617. ACM (2017)

[8] Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Proceedings of FMCAD'18. pp. 1–5. IEEE (2018)

[9] Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Proceedings of CAV'14. LNCS, vol. 8559, pp. 150–166 (2014)

[10] Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Proceedings of CAV'15. LNCS, vol. 9206, pp. 462–469 (2015)

[11] Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Proceedings of ATVA'19. LNCS, vol. 11781, pp. 277–293 (2019)

[12] Abdulla, P.A., Chen, Y.F., Clemente, L., Holík, L., Hong, C.D., Mayr, R., Vojnar, T.: Simulation subsumption in Ramsey-based Büchi automata universality and

inclusion testing. In: Proceedings of CAV'10. LNCS, vol. 6174, pp. 132–147. Springer (2010)

[13] Abdulla, P.A., Chen, Y.F., Clemente, L., Holík, L., Hong, C., Mayr, R., Vojnar, T.: Advanced Ramsey-based Büchi automata inclusion testing. In: Proceedings of CONCUR'11. LNCS, vol. 6901, pp. 187–202. Springer (2011)

[14] Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Proceedings of TACAS'10. LNCS, vol. 6015, pp. 158–174. Springer (2010)

[15] Aickelin, U., Twycross, J., Hesketh-Roberts, T.: Rule generalisation in intrusion detection systems using Snort. International Journal of Electronic Security and Digital Forensics **1**, 101–116 (2008)

[16] Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Proceedings of LICS'18. pp. 46–55. ACM (2018)

[17] Almutairi, A.H., Abdelmajeed, N.T.: Innovative signature based intrusion detection system: Parallel processing and minimized database. In: Proceedings of FADS'17. pp. 114–119. IEEE (2017)

[18] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987)

[19] Assante, M.J., Lee, R.M., Conway, T.: Modular ICS malware. Technical report, Electricity Information Sharing and Analysis Center (E-ISAC) (2017)

[20] Avalle, M., Risso, F., Sisto, R.: Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on GPUs. IEEE/ACM Transactions on Networking **24**(3), 1704–1717 (2016)

[21] Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T., Yu, F.: Parameterized model counting for string and numeric constraints. In: Proceedings of ESEC/FSE'18. pp. 400–410. ACM (2018)

[22] Aziz, A., Balarin, F., Brayton, R.K., Sangiovanni-Vincentelli, A.: Sequential synthesis using S1S. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19**(10), 1149–1162 (2000)

[23] Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Proceedings of CAV'15. LNCS, vol. 9206, pp. 479–486. Springer (2015)

[24] Badr, A.: Hyper-minimization in $\mathcal{O}(n^2)$. In: Proceedings of CIAA'08. LNCS, vol. 5148, pp. 223–231. Springer (2008)

[25] Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)

[26] Baier, C., Kiefer, S., Klein, J., Klüppelholz, S., Müller, D., Worrell, J.: Markov chains and unambiguous Büchi automata. In: Proceedings of CAV'16. LNCS, vol. 9779, pp. 23–42. Springer (2016)

[27] Balajinath, B., Raghavan, S.: Intrusion detection through learning behavior model. Computer Communications **24**(12), 1202–1212 (2001)

[28] Balman, M., Pouyoul, E., Yao, Y., Bethel, E.W., Loring, B., Prabhat, M., Shalf, J., Sim, A., Tierney, B.: Experiences with 100Gbps network applications. In: Proceedings of DIDC'12. pp. 33–42. ACM (2012)

[29] Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. Logical Methods in Computer Science **9**(3), 1–44 (2013)

[30] Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: fast acceleration of symbolic transition systems. In: Proceedings of CAV'03. LNCS, vol. 2725, pp. 118–121. Springer (2003)

[31] Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of CAV'11. LNCS, vol. 6806, pp. 171–177. Springer (2011)

[32] Basin, D., Klarlund, N.: Automata based symbolic reasoning in hardware verification. In: Proceedings of CAV'95. LNCS, vol. 939, pp. 349–361. Springer (1995)

[33] Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S systems to verify parameterized networks. In: Proceedings of TACAS'00. LNCS, vol. 1785, pp. 188–203. Springer (2000)

[34] Beal, M., Crochemore, M.: Minimizing local automata. In: Proceedings of ISIT'07. pp. 1376–1380. IEEE (2007)

[35] Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: Proceedings of CoNEXT'07. pp. 1–12. ACM (2007)

[36] Becchi, M., Crowley, P.: Efficient regular expression evaluation: Theory to practice. In: Proceedings of ANCS'08. pp. 50–59. ACM (2008)

[37] Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: Proceedings of CoNEXT'08. pp. 1–12. ACM (2008)

[38] Becchi, M., Wiseman, C., Crowley, P.: Evaluating regular expression matching engines on network and general purpose processors. In: Proceedings of ANCS'09. pp. 30–39. ACM (2009)

[39] Benedikt, M., Lenhardt, R., Worrell, J.: Model checking Markov chains against unambiguous Büchi automata. CoRR **abs/1405.4560v2** (2016)

[40] Berdine, J.: Private communication (2015)

[41] Berman, L.: The complexity of logical theories. Theoretical Computer Science **11**(1), 71–77 (1980)

[42] Berstel, J.: Transductions and context-free languages. Vieweg+Teubner Verlag (1979)

[43] Berstel, J., Boasson, L., Carton, O., Fagnot, I.: Minimization of automata. CoRR **abs/1010.5318** (2010)

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

[45] Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Proceedings of TACAS'09. LNCS, vol. 5505, pp. 307–321 (2009)

[46] Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Proceedings of CAV'20. LNCS, vol. 12225, pp. 15–27. Springer (2020)

[47] Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.: Complementing semi-deterministic Büchi automata. In: Proceedings of TACAS'16. LNCS, vol. 9636, pp. 770–787. Springer (2016)

[48] Bodeveix, J., Filali, M.: FMona: a tool for expressing validation techniques over infinite state systems. In: Proceedings of TACAS'00. LNCS, vol. 1785, pp. 204–219. Springer (2000)

[49] Boigelot, B.: Lash: Liège automata-based symbolic handler. `https://people.montefiore.uliege.be/boigelot/research/lash/index.html`, [Online; accessed 2021-02-03]

[50] Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Proceedings of IJCAR'01. LNCS, vol. 2083, pp. 611–625. Springer (2001)

[51] Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Proceedings of CAV'94. LNCS, vol. 818, pp. 55–67. Springer (1994)

[52] Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proceedings of POPL'13. pp. 457–468. ACM (2013)

[53] Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Proceedings of CIAA'08. LNCS, vol. 5148, pp. 57–67. Springer (2008)

[54] Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. International Journal on Software Tools for Technology Transfer **14**(2), 167–191 (2012)

[55] Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Proceedings of CAV'04. LNCS, vol. 3114, pp. 372–386. Springer (2004)

[56] Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proceedings of CAV'00. LNCS, vol. 1855, pp. 403–418 (2000)

[57] Boudet, A., Comon, H.: Diophantine equations, Presburger arithmetic and finite automata. In: Proceedings of CAAP'96. LNCS, vol. 1059, pp. 30–43. Springer (1996)

[58] Bozga, M., Iosif, R., Sifakis, J.: Structural invariants for parametric verification of systems with almost linear architectures. CoRR **abs/1902.02696** (2019)

[59] Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Proceedings of CAV'10. LNCS, vol. 6174, pp. 227–242. Springer (2010)

[60] Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Proceedings of FOSSACS'12. LNCS, vol. 7213, pp. 150–164. Springer (2012)

[61] Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Program verification via Craig interpolation for Presburger arithmetic with arrays. In: Proceedings of VERIFY'10. EPiC Series in Computing, vol. 3, pp. 31–46. EasyChair (2012)

[62] Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. In: Proceedings of ISCA'06. pp. 191–202. IEEE (2006)

[63] Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Mathematical theory of Automata, MRI Symposia Series, vol. 12, pp. 529–561. Polytechnic Press, N.Y. (1962)

[64] Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. In: The Collected Works of J. Richard Büchi, pp. 671–683 (1990)

[65] Buchi, 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)

[66] Büchi, J.R.: Weak second-order arithmetic and finite automata. Technical report, The University of Michigan (1959), `http://hdl.handle.net/2027.42/3930`

[67] Bultan, T., Gerber, R., Pugh, W.: Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. ACM Transactions on Programming Languages and Systems **21**(4), 747–789 (1999)

[68] Bustan, D., Grumberg, O.: Simulation based minimization. In: Proceedings of CADE-17. LNCS, vol. 1831, pp. 255–270. Springer (2000)

[69] Câmpeanu, C., Sântean, N., Yu, S.: Minimal cover-automata for finite languages. In: Proceedings of WIA'98. LNCS, vol. 1660, pp. 43–56. Springer (1999)

[70] Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Proceedings of TACAS'08. LNCS, vol. 4963, pp. 428–442. Springer (2008)

[71] Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Proceedings of ICGI'94. LNCS, vol. 862, pp. 139–152. Springer (1994)

[72] Cascarano, N., Rolando, P., Risso, F., Sisto, R.: INFAnt: NFA pattern matching on GPU devices. SIGCOMM Computer Communication Review **40**(5), 20–26 (2010)

[73] Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the ReplaceAll function. In: Proceedings of POPL'17. pp. 1–29. ACM (2018)

[74] Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proceedings of POPL'19 pp. 1–30 (2019)

[75] Chen, Y.F., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Proceedings of APLAS'19. LNCS, vol. 11893, pp. 447–467. Springer (2019)

[76] Chen, Y.F., Havlena, V., Lengál, O., Turrini, A.: A symbolic algorithm for the case-split rule in string constraint solving. In: Proceedings of APLAS'20. LNCS, vol. 12470, pp. 343–363. Springer (2020)

[77] Chen, Y.F., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Proceedings of PLDI'18. pp. 135–150. ACM (2018)

[78] Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming **77**(9), 1006–1036 (2012)

[79] Clark, C.R., Schimmel, D.E.: Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In: Proceedings of FPL'03. LNCS, vol. 2778, pp. 956–959. Springer (2003)

[80] Clemente, L.: Büchi automata can have smaller quotients. In: Proceedings of ICALP'11. LNCS, vol. 6756, pp. 258–270. Springer (2011)

[81] Collier, K.: Major hospital system hit with cyberattack, potentially largest in u.s. history (2020), `https://www.nbcnews.com/tech/security/cyberattack-hits-major-u-s-hospital-system-n1241254`, [Online; Accessed: 2021-02-17]

[82] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2008)

[83] Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and Presburger arithmetic. In: Proceedings of CAV'98. LNCS, vol. 1427, pp. 268–279. Springer (1998)

[84] Cooper, D.C.: Theorem proving in arithmetic without multiplication. Machine intelligence **7**, 91–100 (1972)

[85] Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: Proceedings of SFCS'88. pp. 338–345. IEEE (1988)

[86] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of POPL'78. pp. 84–96. ACM (1978)

[87] Csanky, L.: Fast parallel matrix inversion algorithms. In: Proceedings of SFCS'75. pp. 11–12 (1975)

[88] Cécé, G.: Foundation for a series of efficient simulation algorithms. In: Proceedings of LICS'17. pp. 1–12. IEEE (2017)

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

[90] D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of POPL'14. pp. 541–554. ACM (2014)

[91] D'Antoni, L., Veanes, M.: Monadic second-order logic on finite sequences. In: Proceedings of POPL'17. pp. 232–245. ACM (2017)

[92] David, J.: Average complexity of Moore's and Hopcroft's algorithms. Theoretical Computer Science **417**, 50–65 (2012)

[93] Desharnais, J., Laviolette, F., Tracol, M.: Approximate analysis of probabilistic processes: Logic, simulation and games. In: Proceedings of QEST'18. pp. 264–273. IEEE (2008)

[94] Deza, M.M., Deza, E.: Encyclopedia of Distances. Springer (2009)

[95] Diekert, V.: Makanin's Algorithm, pp. 387–442 (2002)

[96] Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In: Proceedings of CAV'92. LNCS, vol. 575, pp. 255–265. Springer (1992)

[97] Doner, J.E.: Decidability of the weak second-order theory of two successors. Notices of the American Mathematical Society **12** (1965)

[98] Doner, J.: Tree acceptors and some of their applications. Journal of Computer and System Sciences **4**(5), 406–451 (1970)

[99] Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: Proceedings of TACAS'10. LNCS, vol. 6015, pp. 2–22. Springer (2010)

[100] Durand-Gasselin, A., Habermehl, P.: On the use of non-deterministic automata for Presburger arithmetic. In: Proceedings of CONCUR'10. LNCS, vol. 6269, pp. 373–387. Springer (2010)

[101] 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: Proceedings of ATVA'16. LNCS, vol. 9938, pp. 122–129. Springer (2016)

[102] Durnev, V.G., Zetkina, O.V.: On equations in free semigroups with certain constraints on their solutions. Journal of Mathematical Sciences **158**(5), 671–676 (2009)

[103] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Proceedings of CAV'06. LNCS, vol. 4144, pp. 81–94. Springer (2006)

[104] Egly, U.: On the value of antiprenexing. In: Proceedings of LPAR'94. LNCS, vol. 822, pp. 69–83. Springer (1994)

[105] Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: New techniques for WS1S and WS2S. In: Proceedings of CAV'98. LNCS, vol. 1427, pp. 516–520. Springer (1998)

[106] Esparza, J.: Automata Theory: An algorithmic approach (2017)

[107] Etessami, K., Wilke, T., Schuller, R.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM Journal on Computing **34**(5), 1159–1175 (2005)

[108] Etessami, K.: A hierarchy of polynomial-time computable simulations for automata. In: Proceedings of CONCUR'02. LNCS, vol. 2421, pp. 131–144. Springer (2002)

[109] Farwer, B.: omega-automata. In: Automata, Logics, and Infinite Games: A Guide to Current Research. pp. 3–20 (2001)

[110] Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Di Pietro, A.: An improved DFA for fast regular expression matching. SIGCOMM Computer Communication Review **38**(5), 29–40 (2008)

[111] Fiedor, T., Holík, L., Janků, P., Lengál, O., Vojnar, T.: Lazy automata techniques for WS1S. In: Proceedings of TACAS'17. LNCS, vol. 10205, pp. 407–425. Springer (2017)

[112] Fiedor, T., Holík, L., Lengál, O., Vojnar, T.: Nested antichains for WS1S. Acta Informatica **56**(3), 205–228 (2019)

[113] Fischer, M.J., Rabin, M.O.: Super-exponential complexity of Presburger arithmetic. In: Quantifier Elimination and Cylindrical Algebraic Decomposition, pp. 122–135. Springer (1998)

[114] Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of STOC'78. pp. 114–118. ACM (1978)

[115] Friedgut, E., Kupferman, O., Vardi, M.: Büchi complementation made tighter. International Journal of Foundations of Computer Science **17**, 851–868 (2006)

[116] Fritz, C., Wilke, T.: Simulation relations for alternating Büchi automata. Theoretical Computer Science **338**(1), 275–314 (2005)

[117] Fukač, T., Košař, V., Kořenek, J., Matoušek, J.: Increasing throughput of intrusion detection systems by hash-based short string pre-filter. In: Proceedings of LCN'20. pp. 509–514. IEEE (2020)

[118] Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR **abs/1605.09442** (2016)

[119] Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Proceedings of HVC'12. LNCS, vol. 7857, pp. 209–226 (2012)

[120] Gange, G., Ganty, P., Stuckey, P.J.: Fixing the state budget: Approximation of regular languages with small DFAs. In: Proceedings of ATVA'17. LNCS, vol. 10482, pp. 67–83. Springer (2017)

[121] Ganzow, T., Kaiser, L.: New algorithm for weak monadic second-order logic on inductive structures. In: Proceedings of CSL'10. LNCS, vol. 6247, pp. 366–380. Springer (2010)

[122] Gawrychowski, P., Jeż, A.: Hyper-minimisation made efficient. In: Proceedings of MFCS'09. LNCS, vol. 5734, pp. 356–368. Springer (2009)

[123] Gheerbrant, A., Cate, B.t.: Complete axiomatizations of fragments of monadic second-order logic on finite trees. Logical Methods in Computer Science **8**(4) (2012)

[124] Gleick, J.: A bug and a crash: Sometimes a bug is more then a nuisance. `https://around.com/ariane.html` (1996), [Online; Accessed: 2021-04-20]

[125] Glenn, J., Gasarch, W.: Implementing WS1S via finite automata. In: Proceedings of WIA'96. LNCS, vol. 1260, pp. 50–63. Springer (1996)

[126] Godefroid, P.: Using partial orders to improve automatic verification methods. In: Proceedings of CAV'90. LNCS, vol. 531, pp. 176–185. Springer (1990)

[127] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of PLDI'05. pp. 213–223. ACM (2005)

[128] Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Proceedings of SAS'06. LNCS, vol. 4134, pp. 144–160. Springer (2006)

[129] Gonnord, L., Schrammel, P.: Abstract acceleration in linear relation analysis. Science of Computer Programming **93**, 125–153 (2014)

[130] Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of PLDI'11. pp. 62–73. ACM (2011)

[131] Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of PLDI'08. pp. 281–292. ACM (2008)

[132] Gurumurthy, S., Bloem, R., Somenzi, F.: Fair simulation minimization. In: Proceedings of CAV'02. LNCS, vol. 2404, pp. 610–623. Springer (2002)

[133] Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Proceedings of CHARME'03. LNCS, vol. 2860, pp. 96–110. Springer (2003)

[134] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods in System Design **41**(1), 83–106 (2012)

[135] Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: Proceedings of FMCAD'10. pp. 101–109. IEEE (2010)

[136] Hartmanns, A., Wendler, P.: TACAS 2018 artifact evaluation VM. In: Figshare (2018), `https://doi.org/10.6084/m9.figshare.5896615`

[137] Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Automata terms in a lazy WSkS decision procedure. Journal of Automated Reasoning (To Appear)

[138] Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Automata terms in a lazy WSkS decision procedure. In: Proceedings of CADE-27. LNCS, vol. 11716, pp. 300–318. Springer (2019)

[139] Havlena, V., Lengál, O.: Reducing (to) the ranks: Efficient rank-based Büchi automata complementation (technical report). CoRR **abs/2010.07834** (2020)

[140] Havlena, V., Lengál, O.: Reducing (to) the ranks: Efficient rank-based Büchi automata complementation. Submitted to CONCUR'21 (2021)

[141] Havlena, V., Holík, L., Lengál, O., Valeš, O., Vojnar, T.: Antiprenexing for WSkS: A little goes a long way. In: Proceedings of LPAR'20. pp. 298–316. EasyChair (2020)

[142] Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Proceedings of CAV'14. LNCS, vol. 8559, pp. 797–813. Springer (2014)

[143] Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Proceedings of TACAS'95. LNCS, vol. 1019, pp. 89–110. Springer (1995)

[144] Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proceedings of FOCS'95. pp. 453–462. IEEE (1995)

[145] Henzinger, T.A., Kupferman, O., Rajamani, S.K.: Fair simulation. Information and Computation **173**(1), 64–81 (2002)

[146] Hogben, L.: Handbook of Linear Algebra. CRC Press, 2nd edn. (2013)

[147] Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Proceedings of ATVA'12. LNCS, vol. 7561, pp. 187–202. Springer (2012)

[148] Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. In: Proceedings of POPL'18. pp. 1–32. ACM (2018)

[149] Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Technical report (1971)

[150] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

[151] Hsieh, C.L., Vespa, L., Weng, N.: A high-throughput DPI engine on GPU via algorithm/implementation co-optimization. Journal of Parallel and Distributed Computing **88**, 46–56 (2016)

[152] Hune, T., Sandholm, A.: A case study on using automata in control synthesis. In: Proceedings of FASE'00. LNCS, vol. 1783, pp. 349–362. Springer (2000)

[153] Ilie, L., Navarro, G., Yu, S.: On NFA Reductions. In: Theory is Forever, LNCS, vol. 3113, pp. 112–124. Springer (2004)

[154] Ilie, L., Solis-Oba, R., Yu, S.: Reducing the size of NFAs by using equivalences and preorders. In: Proceedings of CPM'05. LNCS, vol. 3537, pp. 310–321. Springer (2005)

[155] Iosif, R., Rogalewicz, A., Šimáček, J.: The tree width of separation logic with recursive definitions. In: Proceedings of CADE-24. LNCS, vol. 7898, pp. 21–38. Springer (2013)

[156] Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K.: Kargus: A highly-scalable software-based intrusion detection system. In: Proceedings of CCS'12. pp. 317–328. ACM (2012)

[157] Jensen, J.L., Jørgensen, M.E., Klarlund, N., Schwartzbach, M.I.: Automatic verification of pointer programs using monadic second-order logic. SIGPLAN Notices **32**(5), 226–234 (1997)

[158] Jeundefined, A.: Recompression: A simple and powerful technique for word equations. Journal of the ACM **63**(1) (2016)

[159] Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM Journal on Computing **22**(6), 1117–1141 (1993)

[160] Jin Kim, Nara Shin, Jo, S.Y., Sang Hyun Kim: Method of intrusion detection using deep neural network. In: Proceedings of BIGCOMP'17. pp. 313–316. IEEE (2017)

[161] Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Proceedings of ICALP'08. LNCS, vol. 5125, pp. 724–735. Springer (2008)

[162] Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science **134**(2), 329–363 (1994)

[163] Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Proceedings of ATVA'09. LNCS, vol. 5799, pp. 228–243. Springer (2009)

[164] Kaštil, J., Kořenek, J.: Hardware accelerated pattern matching based on deterministic finite automata with perfect hashing. In: Proceedings of DDECS'10. pp. 149–152. IEEE (2010)

[165] Kaštil, J., Kořenek, J.: High speed pattern matching algorithm based on deterministic finite automata with faulty transition table. In: Proceedings of ANCS'10. pp. 1–2. IEEE (2010)

[166] Kaštil, J., Kořenek, J., Lengál, O.: Methodology for fast pattern matching by deterministic finite automaton with perfect hashing. In: Proceedings of DSD'09. pp. 823–829 (2009)

[167] Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology **21**(4), 25:1–25:28 (2012)

[168] King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)

[169] Klaedtke, F.: Bounds on the automata size for Presburger arithmetic. ACM Transactions on Computational Logic **9**(2) (2008)

[170] Klarlund, N., Nielsen, M., Sunesen, K.: A case study in automated verification based on trace abstractions. In: Formal System Specification, LNCS, vol. 1169, pp. 341–373. Springer (1996)

[171] Klarlund, N.: A theory of restrictions for logics and automata. In: Proceedings of CAV'99. LNCS, vol. 1633, pp. 406–417. Springer (1999)

[172] Klarlund, N., Møller, A.: MONA Version 1.4 User Manual (2001), `http://www.brics.dk/mona/`, revision of BRICS NS-98-3

[173] Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. International Journal of Foundations of Computer Science **13**(4), 571–586 (2002)

[174] Klarlund, N., Nielsen, M., Sunesen, K.: Automated logical verification based on trace abstractions. In: Proceedings of PODC'96. pp. 101–110. ACM (1996)

[175] Klein, J., Baier, C.: On-the-fly stuttering in the construction of deterministic ω-automata. In: Proceedings of CIAA'07. LNCS, vol. 4783, pp. 51–61. Springer (2007)

[176] Körner, H.: On minimizing cover automata for finite languages in $\mathcal{O}(n \log n)$ time. In: Proceedings of CIAA'02. LNCS, vol. 2608, pp. 117–127. Springer (2002)

[177] Kořenek, J., Kobierský, P.: Intrusion detection system intended for multigigabit networks. In: Proceedings of DDECS'07. pp. 1–4. IEEE (2007)

[178] Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, 1st edn. (2008)

[179] Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: Proceedings of ANCS'07. pp. 155–164. ACM, New York, NY, USA (2007)

[180] Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: Proceedings of SIGCOMM'06. pp. 339–350. ACM (2006)

[181] Kumar, S., Turner, J.S., Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. In: Proceedings of ANCS'06. pp. 81–92. ACM (2006)

[182] Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Transactions on Computational Logic **2**(3), 408–429 (2001)

[183] Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. Journal of Computer and System Sciences **35**(1), 59–71 (1987)

[184] Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Proceedings of APLAS'18. LNCS, vol. 11275, pp. 350–372 (2018)

[185] Levi, F.W.: On semigroups. Bulletin of the Calcutta Mathematical Society **36**, 141–146 (1944)

[186] Li, Y., Sun, X., Turrini, A., Chen, Y.F., Xu, J.: ROLL 1.0: ω-regular language learning library. In: Proceedings of TACAS'19. LNCS, vol. 11427, pp. 365–371. Springer (2019)

[187] Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Proceedings of VMCAI'18. LNCS, vol. 10747, pp. 313–335. Springer (2018)

[188] Li, Y., Vardi, M.Y., Zhang, L.: On the power of unambiguity in Büchi complementation. In: Proceedings of GandALF'20. EPTCS, vol. 326, pp. 182–198. Open Publishing Association (2020)

[189] Liang, T., Reynolds, A., Tinelli, C., Barrett, C.W., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Proceedings of CAV'14. LNCS, vol. 8559, pp. 646–662 (2014)

[190] Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Proceedings of POPL'16. pp. 123–136. ACM (2016)

[191] Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: Proceedings of ATVA'18. LNCS, vol. 11138, pp. 352–369 (2018)

[192] Lin, C., Huang, C., Jiang, C., Chang, S.: Optimization of pattern matching circuits for regular expression on FPGA. IEEE Transactions on Very Large Scale Integration Systems **15**(12), 1303–1310 (2007)

[193] Lin, C., Liu, C., Chang, S.: Accelerating regular expression matching using hierarchical parallel machines on GPU. In: Proceedings of GLOBECOM'11. pp. 1–5. IEEE (2011)

[194] Luchaup, D., De Carli, L., Jha, S., Bach, E.: Deep packet inspection with DFA-trees and parametrized language overapproximation. In: Proceedings of INFOCOM'14. pp. 531–539. IEEE (2014)

[195] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proceedings of POPL'11. pp. 611–622. ACM (2011)

[196] Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Proceedings of SAS'11. LNCS, vol. 6887, pp. 43–59. Springer (2011)

[197] Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **145**(2), 147–236 (1977)

[198] Maletti, A., Quernheim, D.: Optimal hyper-minimization. International Journal of Foundations of Computer Science **22**(8), 1877–1891 (2011)

[199] Margaria, T., Steffen, B., Topnik, C.: Second-order value numbering. In: Proceedings of GraMoT'10. ECEASST, vol. 30, pp. 1–15. EASST (2010)

[200] Matiyasevich, Y.: Computation paradigms in light of Hilbert's tenth problem. In: New computational paradigms, pp. 59–85. Springer (2008)

[201] Matiyasevich, Y.V.: A connection between systems of word and length equations and Hilbert's tenth problem. Zapiski Nauchnykh Seminarov POMI **8**, 132–144 (1968)

[202] Matoušek, D., Kořenek, J., Puš, V.: High-speed regular expression matching with pipelined automata. In: Proceedings of FPT'16. pp. 93–100. IEEE (2016)

[203] Matoušek, P., Ryšavý, O., Havlena, V., Grégr, M.: Flow based monitoring of ICS communication in the smart grid. Journal of Information Security and Applications **2020**(54), 1–16 (2020)

[204] Matoušek, D., Kořenek, J., Puš, V.: High-speed regular expression matching with pipelined automata. In: Proceedings of FPT'16. pp. 93–100. IEEE (2016)

[205] Matt Jonkman et al.: Suricata (2017), `http://suricata-ids.org`

[206] Mayr, R., Clemente, L.: Advanced automata minimization. In: Proceedings of POPL'13. pp. 63–74. ACM (2013)

[207] Mayr, R., Clemente, L.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Logical Methods in Computer Science **15**, 1–73 (2019)

[208] Mayr, R., et al.: Reduce 2.4.5: A tool for minimizing nondeterministic finite-word and Büchi automata. `http://languageinclusion.org/doku.php?id=tools`, [Online; accessed 2017-09-30]

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

[210] Mealy, G.H.: A method for synthesizing sequential circuits. Bell System Technical Journal **34**(5), 1045–1079 (1955)

[211] Michel, M.: Complementation is more difficult with automata on infinite words. CNET, Paris **15** (1988)

[212] Mohamed, A.B., Idris, N.B., Shanmugum, B.: A brief introduction to intrusion detection system. In: Proceedings of IRAM'12. CCIC, vol. 330, pp. 263–271. Springer (2012)

[213] Mohri, M.: Edit-distance of weighted automata. In: Proceedings of CIAA'02. LNCS, vol. 2608, pp. 1–23. Springer (2002)

[214] Mohri, M.: A disambiguation algorithm for finite automata and functional transducers. In: Proceedings of CIAA'12, LNCS, vol. 7381, pp. 265–277. Springer (2012)

[215] Møller, A., Schwartzbach, M.: The pointer assertion logic engine. In: Proceedings of PLDI'01. pp. 221–231. ACM (2001)

[216] Moore, F.E.: Gedanken-experiments on sequential machines. Annals of Mathematics studies **34**, 129–153 (1956)

[217] Morawietz, F., Cornell, T.: The MSO logic-automaton connection in linguistics. In: Proceedings of LACL'97, LNAI, vol. 1582, pp. 112–131. Springer (1997)

[218] de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)

[219] Nielsen, J.: Die isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Mathematische Annalen **78**(1), 385–397 (1917)

[220] O'Kane, S.: Boeing finds another software problem on the 737 Max. `https://www.theverge.com/2020/2/6/21126364/boeing-737-max-software-glitch-flaw-problem` (2020), [Online; Accessed: 2021-04-20]

[221] Oppen, D.C.: A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. Journal of Computer and System Sciences **16**(3), 323–332 (1978)

[222] Osera, P.M.: Constraint-based type-directed program synthesis. In: Proceedings of TyDe'19. pp. 64–76. ACM (2019)

[223] Papadimitriou, C.M.: Computational complexity. Addison-Wesley (1994)

[224] Parker, A.J., Yancey, K.B., Yancey, M.P.: Regular language distance and entropy. CoRR **abs/1602.07715** (2016)

[225] Păun, A., Sântean, N., Yu, S.: An $\mathcal{O}(n^2)$ algorithm for constructing minimal cover automata for finite languages. In: Proceedings of CIAA'00. LNCS, vol. 2088, pp. 243–251. Springer (2001)

[226] Peled, D.A.: All from one, one for all: on model checking using representatives. In: Proceedings of CAV'93. LNCS, vol. 697, pp. 409–423. Springer (1993)

[227] Perrin, D., Pin, J.: Infinite words: Automata, Semigroups, Logic and Games. Pure and Applied Mathematics, Elsevier (2004)

[228] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proceedings of LICS'06. pp. 255–264. IEEE (2006)

[229] Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: Proceedings of FOCS'99. pp. 495–500. IEEE (1999)

[230] Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of STOC'06. pp. 467–476. ACM (2006)

[231] Presburger, M.: Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In: Comptes Rendus du I congres de Mathe- maticiens des Pays Slaves. pp. 92–101 (1929)

[232] Pugh, W., Wonnacott, D.: Constraint-based array dependence analysis. ACM Transactions on Programming Languages and Systems **20**(3), 635–678 (1998)

[233] Pugh, W., et al.: The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. `http://www.cs.umd.edu/projects/omega/`, [Online; accessed 2021-02-04]

[234] Puš, V., Tobola, J., Košař, V., Kaštil, J., Kořenek, J.: Netbench: Framework for evaluation of packet processing algorithms. Proceedings of ANCS'11 pp. 95–96 (2011)

[235] Quine, W.V.: Concatenation as a basis for arithmetic. The Journal of Symbolic Logic **11**(4), 105–114 (1946)

[236] Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development **3**(2), 114–125 (1959)

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

[238] Ranzato, F., Tapparo, F.: A new efficient simulation equivalence algorithm. In: Proceedings of LICS'07. pp. 171–180. IEEE (2007)

[239] Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Information and Computation **208**(1), 1–22 (2010)

[240] Rawlings, J.O., Pantula, S.G., Dickey, D.A.: Applied Regression Analysis: A Research Tool. Springer, New York, 2nd edn. (1998)

[241] Redziejowski, R.: An improved construction of deterministic omega-automaton using derivatives. Fundamenta Informaticae **119**, 393–406 (2012)

[242] Reinhardt, K.: The Complexity of Translating Logic to Finite Automata, pp. 231–238. Springer (2002)

[243] Revuz, D.: Minimisation of acyclic deterministic automata in linear time. Theoretical Computer Science **92**(1), 181–189 (1992)

[244] Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Proceedings of CAV'17. LNCS, vol. 10427, pp. 453–474 (2017)

[245] Robson, J.M., Diekert, V.: On quadratic word equations. In: Proceedings of STACS'99. LNCS, vol. 1563, pp. 217–226. Springer (1999)

[246] Russel, R., et al.: Netfilter. `http://netfilter.org`

[247] Sadegh, G.: Complementing Büchi automata. Technical report, Laboratoire de Recherche et Développement de l'Epita (2009)

[248] Safra, S.: On the complexity of ω-automata. In: Proceedings of FOCS'88. pp. 319–327. IEEE (1988)

[249] Sandholm, A., Schwartzbach, M.I.: Distributed safety controllers for web services. In: Proceedings of FASE'98. LNCS, vol. 1382, pp. 270–284. Springer (1998)

[250] Schewe, S.: Büchi complementation made tight. In: Proceedings of STACS'09. pp. 661–672. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2009)

[251] Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Proceedings of IWWERT'90. LNCS, vol. 572, pp. 85–150. Springer (1990)

[252] Shashank, K., Balachandra, M.: Review on network intrusion detection techniques using machine learning. In: Proceedings of DISCOVER'18. pp. 104–109. IEEE (2018)

[253] Shützenberger, M.: On the definition of a family of automata. Information and Control **4**(2), 245–270 (1961)

[254] Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proceedings of FCCM'01. pp. 227–238. IEEE (2001)

[255] 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)

[256] Smith, M.A., Klarlund, N.: Verification of a sliding window protocol using IOA and MONA. In: Proceedings of FORTE/PSTV'00. IFIP, vol. 183, pp. 19–34. Kluwer (2000)

[257] Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: Proceedings of SP'08. pp. 187–201. IEEE (2008)

[258] Solodovnikov, V.I.: Upper bounds on the complexity of solving systems of linear equations. Journal of Soviet Mathematics **29**(4), 1482–1501 (1985)

[259] Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Proceedings of CAV'00. LNCS, vol. 1855, pp. 248–263. Springer (2000)

[260] Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: Proceedings of CCS'03. pp. 262–271. ACM (2003)

[261] Sourdis, I., Bispo, J., Cardoso, J., Vassiliadis, S.: Regular expression matching in reconfigurable hardware. Signal Processing Systems **51**, 99–121 (2008)

[262] Spencer, H.: A Regular-Expression Matcher, pp. 35–71. Academic Press Professional, Inc., USA (1994)

[263] StackStatus: Outage postmortem - july 20, 2016. `https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016` (2016), [Online; Accessed: 2021-04-20]

[264] Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Fifth Annual ACM Symposium on Theory of Computing. pp. 1–9. Proceedings of STOC'73, ACM (1973)

[265] Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Proceedings of LPAR'05. LNCS, vol. 3835, pp. 396–411. Springer (2005)

[266] Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951)

[267] Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Transactions on Software Engineering and Methodology **22**(4), 33:1–33:33 (2013)

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

[269] The Snort Team: Snort, `http://www.snort.org`

[270] Thollard, F., Clark, A.: Learning stochastic deterministic regular languages. In: Proceedings of ICGI'04. LNCS, vol. 3264, pp. 248–259. Springer (2004)

[271] Thompson, K.: Programming techniques: Regular expression search algorithm. Communications of the ACM **11**(6), 419–422 (1968)

[272] Topnik, C., Wilhelm, E., Margaria, T., Steffen, B.: jMosel: a stand-alone tool and jabc plugin for M2L(Str). In: Proceedings of SPIN'06. LNCS, vol. 3925, pp. 293–298. Springer (2006)

[273] Traytel, D.: A coalgebraic decision procedure for WS1S. In: Proceedings of CSL'15. LIPIcs, vol. 41, pp. 487–503. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015)

[274] Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of CCS'14. pp. 1232–1243. ACM (2014)

[275] Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Proceedings of CAV'16. LNCS, vol. 9779, pp. 218–240. Springer (2016)

[276] Tsai, M.H., Fogarty, S., Vardi, M.Y., Tsay, Y.K.: State of Büchi complementation. In: Proceedings of CIAA'10. LNCS, vol. 6482, pp. 261–271. Springer (2011)

[277] Tsai, M.H., Tsay, Y.K., Hwang, Y.S.: GOAL for games, omega-automata, and logics. In: Proceedings of CAV'13. LNCS, vol. 8044, pp. 883–889. Springer (2013)

[278] Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In: Proceedings of RAID'07. LNCS, vol. 4637, pp. 107–126 (2007)

[279] Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings of ICATPN'89. LNCS, vol. 483, pp. 491–515. Springer (1991)

[280] Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. pp. 327–338. Proceedings of SFCS'85, IEEE (1985)

[281] Vardi, M.Y., Wilke, T.: Automata: From logics to algorithms. Logic and Automata **2**, 629–736 (2008)

[282] 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`

[283] 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, pp. 155–175. Springer (2018)

[284] Vern Paxson et al.: The Bro Network Security Monitor (2018), http://www.bro.org

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

[286] Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: Proceedings of CAV'16. LNCS, vol. 9779, pp. 241–260. Springer (2016)

[287] Wang, L., Chen, S., Tang, Y., Su, J.: Gregex: GPU based high speed regular expression matching engine. In: Proceedings of IMIS'11. pp. 366–370. IEEE (2011)

[288] Wang, Y., Zhou, M., Jiang, Y., Song, X., Gu, M., Sun, J.: A static analysis tool with optimizations for reachability determination. In: Proceedings of ASE'17. pp. 925–930. IEEE (2017)

[289] Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Proceedings of CADE-23. LNCS, vol. 6803, pp. 476–491. Springer (2011)

[290] Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints. In: Proceedings of SAS'95. LNCS, vol. 983, pp. 21–32. Springer (1995)

[291] Wulf, M.D., Doyen, L., Raskin, J.F.: A lattice theory for solving games of imperfect information. In: Proceedings of HSCC'06, LNCS, vol. 3927, pp. 153–168. Springer (2006)

[292] Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Proceedings of CAV'06. LNCS, vol. 4144, pp. 17–30. Springer (2006)

[293] Wulf, M.D., Doyen, L., Maquet, N., Raskin, J.: Antichains: Alternative algorithms for LTL satisfiability and model-checking. In: Proceedings of TACAS'08. LNCS, vol. 4963, pp. 63–77 (2008)

[294] Yan, Q.: Lower bounds for complementation of ω-automata via the full automata technique. In: Proceedings of ICALP'06. LNCS, vol. 4052, pp. 589–600. Springer (2006)

[295] Yang, J., Jiang, L., Bai, X., Peng, H., Dai, Q.: A high-performance round-robin regular expression matching architecture based on FPGA. In: Proceedings of ISCC'18. pp. 1–7. IEEE (2018)

[296] Yang, Y., Prasanna, V.: High-performance and compact architecture for regular expression matching on FPGA. IEEE Transactions on Computers **61**(7), 1013–1025 (2012)

[297] Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: Proceedings of TACAS'10. LNCS, vol. 6015, pp. 154–157. Springer (2010)

[298] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1), 44–70 (2014)

[299] Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: Proceedings of ANCS'06. pp. 93–102. ACM (2006)

[300] Yu, F., Shueh, C.Y., Lin, C.H., Chen, Y.F., Wang, B.Y., Bultan, T.: Optimal sanitization synthesis for web application vulnerability repair. In: Proceedings of ISSTA'16. pp. 189–200. ACM (2016)

[301] Yun, S., Lee, K.: Optimization of regular expression pattern matching circuit using at-most two-hot encoding on FPGA. In: Proceedings of FLP'10. pp. 40–43. IEEE (2010)

[302] Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: Proceedings of POPL'08. pp. 349–361. ACM (2008)

[303] Zhao, Z., Sadok, H., Atre, N., Hoe, J.C., Sekar, V., Sherry, J.: Achieving 100Gbps intrusion prevention on a single server. In: Proceedings of OSDI'20. pp. 1083–1100. USENIX Association (2020)

[304] Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design **50**(2-3), 249–288 (2017)

[305] Zhou, M., He, F., Wang, B., Gu, M., Sun, J.: Array theory of bounded elements and its applications. Journal of Automated Reasoning **52**(4), 379–405 (2014)

[306] Češka, M., Havlena, V., Holík, L., Kořenek, J., Lengál, O., Matoušek, D., Matoušek, J., Semrič, J., Vojnar, T.: Deep packet inspection in FPGAs via approximate nondeterministic automata. In: Proceedings of FCCM'19. pp. 109–117. IEEE (2019)

[307] Češka, M., Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Approximate reduction of finite automata for high-speed network intrusion detection. International Journal on Software Tools for Technology Transfer (22), 523–539 (2019)