BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EFFICIENT ALGORITHMS FOR COUNTING AUTOMATA

EFEKTIVNÍ ALGORITMY PRO PRÁCI S ČÍTACÍMI AUTOMATY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **DAVID MIKŠANÍK**
AUTOR PRÁCE

SUPERVISOR — **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

23029

Student: **Mikšaník David**
Programme: Information Technology
Title: **Efficient Algorithms for Counting Automata**
Category: Theoretical Computer Science
Assignment:

1. Learn the theory of symbolic finite automata and counting automata.
2. Study algorithms for efficient handling of finite and symbolic automata, in particular algorithms for efficient computation of intersection of two automata, testing universality, language inclusion, and the simulation relation.
3. Design efficient algorithms for selected operations over counting automata.
4. Implement the designed algorithms and experimentally evaluate their performance.
5. Discuss the obtained results.

Recommended literature:

- L. Holik, O. Lengal, O. Saarikivi, L. Turonova, M. Veanes, and T. Vojnar. Succinct Determinisation of Counting Automata via Sphere Construction. In *Proc. of APLAS'19.* 2019. Springer.
- P.A. Abdulla, Y. Chen, L. Holik, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (on Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158--174, 2010. Springer.
- Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14). ACM, New York, NY, USA, 541-553.

Requirements for the first semester:

- First two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Lengál Ondřej, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: January 13, 2020

Abstract

Counting automata (CAs) are classical finite automata extended with bounded counters. They still denote the class of regular languages but in a more compact way than finite automata. Since CAs are a recent model, there is a gap in the knowledge of efficient algorithms implementing various operations on the CAs. In this thesis, we mainly focus on an existing subclass of CAs called monadic counting automata (MCAs), i.e., CAs with counting loops on character classes, which are common in practice (e.g., detection of packets in network traffic, log analysis). For this subclass, we efficiently solve the emptiness and inclusion problems. Moreover, we provide two extensions of the class of MCAs (but not beyond the class of CAs) and efficiently solve the emptiness problem for them. MCAs naturally arise from regular expressions that are extended by the counting operator limited only to character classes. Thus our algorithm solving the inclusion problem of MCAs can be used in a new method for solving the inclusion problem of such regular expressions. We experimentally evaluated this method on regular expressions from a wide range of applications and compared it with the naive method. The experiments show that the method using our algorithm is less prone the explode. It also outperforms the naive method if the regular expressions contain counting operators with large bounds. As expected, for the easy cases, the naive method is still faster than the method based on our algorithm.

Abstrakt

Čítací automaty (CA) jsou klasické konečné automaty rozšířené o omezené čítače. CA stále reprezentují třídu regulárních jazyků, ale kompaktněji než konečné automaty. Jelikož jsou CA nedávným modelem, chybějí zde efektivní algoritmy implementující různé operace nad nimi. V této práci se primárně soustředíme na existující podtřídu CA zvanou monadické čítací automaty (MCA). Jsou to CA s čítacími smyčkami na třídě znaků, které se často vyskytují v praxi (např. při detekci paketů v síťovém provozu nebo analýze log souborů). Pro tuto podtřídu efektivně vyřešíme problémy prázdnosti a inkluze. Navíc poskytneme dvě rozšíření třídy MCA, které jsou stále podtřídou CA, a vyřešíme pro ně efektivně problém prázdnosti. MCA přirozeně vznikají z regulárních výrazů, které jsou rozšířené o čítací operátory vyskytující se pouze na třídě znaků. Náš algoritmus řešící problém inkluze MCA tedy může být použit jako základ nové metody pro testování inkluze takových regulárních výrazů. Tento přístup jsme experimentálně vyhodnotili na regulárních výrazech z praxe a porovnali s naivní metodou. Experimenty ukazují, že metoda používající náš algoritmus je více odolná proti stavové explozi. Také překonává naivní metodu, pokud regulární výrazy obsahují čítací operátory s velkými mezemi. Podle očekávání, pro jednoduché případy je naivní metoda stále rychlejší než metoda používající náš algoritmus.

Keywords

finite automata, counting automata, emptiness problem, inclusion, regular expressions

Klíčová slova

konečné automaty, čítací automaty, problém prázdnosti, inkluze, regulární výrazy

Reference

MIKŠANÍK, David. *Efficient Algorithms for Counting Automata.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Čítací automaty (CA) jsou nedávným modelem pro reprezentaci třídy regulárních jazyků [14]. Můžeme si představit, že CA jsou klasické konečné automaty rozšířené o omezené čítače (tj. každý čítač může nabývat konečně mnoha hodnot). Poté přechody mezi stavy nezávisí pouze na vstupním symbolu, ale také na aktuální konfiguraci čítačů a jestli CA vstupní slovo přijme nezávisí pouze na tom, v jakém stavu skončíme, ale také na koncové konfiguraci čítačů. Pro úplnost, iniciální stav čítacího automatu není dán pouze počátečními stavy, ale i počáteční konfigurací čítačů. Jedna z motivací pro zavedení CA je redukce počtu stavů (tedy i přechodů) v nedeterministických konečných automatech (NFA). Z toho plyne výhoda CA proti NFA—CA kompaktněji reprezentují třídu regulárních jazyků. V literatuře existují modely, které se snaží pouze o redukci počtu přechodů v NFA, například symbolické konečné automaty [13, 23] redukují přechody efektivnějším způsobem než CA. Upozorňujeme, že lze jednoduše rozšířit definici CA tak, aby poskytovala všechny výhody symbolických automatů. Použití takového automatu potom vede na efektivnější redukci přechodů než za použití CA. Jednu možnou definici takového automatu poskytneme v naší práci.

Každý čítač v CA nabývá pouze konečně mnoha hodnot, tedy počet všech možných konfigurací čítačů je konečný. Proto jednoduše každý CA může být převeden na ekvivalentní NFA tak, že se každá konfigurace čítačů zakóduje do jednoho stavu NFA. Z toho plyne, že každá operace nad CA (sjednocení, průnik, atd.) může být převedena na operaci nad NFA. Takový postup je možný, ale neefektivní, protože časová složitost takto řešených operací je potom stejná jako časová složitost operací nad NFA. Poznamenejme, že není známý žádný efektivní algoritmus řešící různé operace nad CA kromě determinizačního algoritmu [14]. Existence efektivního determinizačního algoritmu naznačuje, že je možné takové efektivní algoritmy implementující různé operace nad CA vytvořit. V této práci se návrhu daných algoritmů věnuji.

Zejména se soustředíme na existující podtřídu CA, na tzv. monadické čítací automaty (MCA)—čítací automaty kde dochází k inkrementaci čítače pouze na smyčkách (přechod, který začíná a končí ve stejném stavu) jednotlivých stavů. MCA se často vyskytují v praxi (např. při detekci paketů v síťovém provozu nebo analýze log souborů). MCA přirozeně reprezentují *rozšířené regulární výrazy* (dále jen regulární výrazy), jsou to standardní regulární výrazy rozšířené o počítání na skupině (třídě) znaků. Takové regulární výrazy stále značí třídu regulárních jazyků, ale kompaktněji než standardní regulární výrazy (např. `[abc]{5}` značí všechny řetězce délky 5 kde každý symbol je buď `a`, `b`, nebo `c`). Autoři CA poskytují efektivnější determinizační algoritmus pokud vstupem je MCA.

Vymysleli jsme efektivní řešení (algoritmus) pro testovaní jazykové inkluze MCA, které imituje řešení pro testování jazykové inkluze NFA—sestavíme produkt automat ze vstupních dvou NFA a hledáme v něm dosažitelné koncové stavy. Aby jsme tenhle postup mohli aplikovat i pro MCA M_1 a M_2, museli jsme najít odpovědi na následující problémy: jak vypočítat komplement automatu M_2, jak sestavit produkt automat $M_1 \times \overline{M_2}$ automatu M_1 a komplementu M_2, a jak efektivně určit zda stav v produkt automatu je dosažitelný. Také jsme rozšířili třídu MCA na dvě nové podtřídy CA. Jak ukážeme na příkladech, tyto nové podtřídy dokáží reprezentovat komplexnější regulární výrazy. Například nejsme limitovaní pouze na počítaní na skupině znaků, ale může přímo počítat sekvence znaků (např. `(abc){5}` značí řetězec, který vznikne konkatenací řetězce `abc` 5 krát za sebou). Pro tyto podtřídy (včetně MCA) jsme efektivně vyřešili problém prázdnosti. Mimo jiné jsme intuitivně ukázali proč řešení problémy prázdnosti a inkluze obecných CA vyžadují použití NFA.

Existence našeho algoritmu pro řešení inkluze MCA otevírá novou možnost jak testovat jazykovou inkluzi regulárních výrazů—vstupní regulární výrazy převedeme na MCA a poté aplikujeme náš algoritmus. Implementovali jsme náš algoritmus pro testovaní jazykové inkluze MCA a využili jsme knihovnu Automata od Microsoftu [3], která poskytuje prostředky pro převod regulárních výrazů na MCA a determinizační algoritmus pro MCA. Tento přístup jsme experimentálně ověřili na regulárních výrazech z praxe a porovnali s naivní metodou, která je založena na převodu regulárních výrazu na NFA, implementovanou v [1]. Přestože náš algoritmus není optimalizovaný a chybí implementace jedné akcelerační formule pro smyčky v deterministickém MCA, experimenty ukazují, že metoda používající náš algoritmus je více odolná proti stavové explozi. Zejména se jedná o regulární výrazy použité v Bro [22]. Pokud vstupem jsou regulární výrazy s čítacím operátory, které mají velké meze, tak metoda založena na MCA překonává naivní metodu. Pro jednoduché regulární výrazy (kde regulární výrazy obsahují 1.6 čítacích operátorů s mezí 110 v průměru) je naivní metoda očekávaně rychlejší než metoda založená na MCA.

V naší implementaci algoritmu pro řešení inkluze MCA používáme Z3 SMT solver [4] s lineární celočíselnou logikou pro práci s formulemi. Připomínáme, že nejsme schopni v této logice implementovat jednu akcelerační formuli pro smyčky v determinizovaném MCA. Existence takové implementace zcela jistě dále zvýší výkonnost naše algoritmu. Mimo jiné v naší implementaci je prostor pro vyzkoušení jiných (efektivnějších) algeber pro reprezentaci symbolu v (determinizovaném) MCA. Vidíme také možnost integrace našeho algoritmu do knihovny Automata od Microsoftu [3], která už poskytuje nějaké prostředky práci s MCA.

Stejné metoda pro určování dosažitelných stavu v produkt automatu $M_1 \times \overline{M_2}$ může být použita pro minimalizaci deterministický MCA, které vzniknou aplikací determinizačního algoritmu v [14, Sekce 4.2], ve smyslu odstranění nedosažitelných stavů. Myslíme si, že tato metoda může být dále upravena, tak aby byla přímo součástí výše uvedeného determinizačního algoritmu. Důsledkem by bylo, že by algoritmus negeneroval nedosažitelné stavy, které doposud může generovat (viz [14]).

Naše řešení pro inkluzi MCA přesně reprezentuje stavy z tzv. subset konstrukce. Jako další rozšíření práce lze uvažovat aplikace subsumpce pro prořezávání množiny dosažených stavů, např. na podobném principu jako používá algoritmus Antichains pro testování inkluze NFA. Toto spočívá v zamezení prozkoumání stavů, jejichž sémantika je z hlediska testování inkluze pokryta sémantikou dosažených stavů.

Efficient Algorithms for Counting Automata

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of 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.

. . . .
David Mikšaník
May 28, 2020

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D for his guidance, feedback, and help with this thesis. Furthermore, I would like to thank Ing. Lenka Turoňová for her help with Automata and Transducer Library for .NET.

Contents

Chapter 1

Introduction

Classical finite automata (FAs) together with regular expressions (REs) are the main models for describing the class of regular languages. Usually, in the computers, REs are represented by the FAs. Thus the applications of FAs are at least as wide as the applications of REs. For example, in the text processing (searching in logs, detecting packets in the network traffic), compilers (lexical and syntax analyzer), or formal verification to name a few. Although FAs are working on a finite state space with a finite alphabet by the definition, sometimes FAs are too large to be stored in computers. Suppose that an FA N implements some RE r (i.e., N denotes the same language as r). If symbols in r are encoded by ASCII or UTF-16, then the alphabet has 2^8 or 2^{16} symbols, respectively. Hence the number of transitions from each state in N is either 2^8 or 2^{16}. This example shows one disadvantage of FAs—they do not scale well with the growing number of symbols in the alphabets and if the alphabet is infinite, then it is impossible to use FAs.

There are several techniques to reduce the number of transitions in FAs (e.g., partial transition relation to avoid irrelevant symbols). But a radical reduction in the number of transitions comes from the use of symbolic finite automata (SFAs) [13, 23], i.e., FAs where each transition is annotated by a predicate that denotes a set (possibly infinity) of symbols. The advantage of SFAs over FAs is that multiple transitions between the same states in FAs can be represented in SFAs by a single transition. Nevertheless, SFAs like FAs still suffer from the state explosion (e.g., in the determinization of FAs or SFAs). Consider the extended regular expression (eRE)[1] `.*a.{k}` for $\mathtt{k} \geq 0$, then the smallest equivalent deterministic finite or symbolic automaton has $2^{\mathtt{k}+1}$ states. Even for relatively small values of `k`, the resulting deterministic automaton has so many states that it is impossible to store such an automaton in any computer.

In the literature, there are also several automata models that are designed for the reduction in the number of states (e.g., [17, 21]). In this thesis, we focus on one recent model, the so-called counting automata (CAs) [14]. All these models represent eREs in a succinct way, but [14] provides also an efficient determinization algorithm of CAs. For instance, the smallest equivalent CA for the eRE `.*a.{k}` has two states regardless of the value of `k` (in contrast to nondeterministic FAs (NFAs), where the size depends linearly on `k`) and the smallest equivalent deterministic CA has only $\mathtt{k} + 2$ states (cf. [14]). Therefore, using this model we often significantly reduce the number of states compared to the equivalent (deterministic) FAs.

[1] Until we give a precise definition of extended regular expression, we can use POSIX extended regular expressions that still denote the class of regular language, but in a more succinct way than the standard regular expression.

We note that any CA N can be transformed to an equivalent NFA N' by unfolding every possible configuration in N into part of states in N'. Therefore, any operation on CAs (union, intersection, etc.) can be transformed in the terms of NFAs. Such solutions are not the most efficient ones (but perhaps the only ones possible), since the time complexity of the algorithms performing the operations remains the same as for NFAs. To the best of our knowledge, there are no known efficient algorithms implementing various operations on CAs except the determinization algorithm. The existence of the determinization algorithm of CAs suggests that it is possible to find efficient algorithms performing operations on CAs. In this thesis, we give a partial answer by designing algorithms for some operations on CAs.

We restrict ourselves mainly to the subclass of CAs that are common in practice. This subclass is called monadic counting automata (MCAs), i.e., CAs with counting loops on character class. MCAs naturally arise from eREs, where counting is limited to the character classes (e.g., `[abc]{0,5}` or `.*a.{k}`). We note that [14] provides an even more efficient determinization algorithm if the input CA is also an MCA. For this subclass, we give an efficient solution (algorithm) to the language inclusion problem in a similar manner as for NFAs—we build the product automaton of MCAs and search for a reachable final state. Such an approach is straightforward but not as easy as for NFAs, because the next move of CAs does not depend only on the input symbol but also on the actual configuration of counters. Moreover, we introduce two new subclasses of CAs, which are both larger than MCAs. For all these subclasses (including MCAs) we give an efficient solution to the emptiness problem. Besides the main work, we also give an intuition about why the emptiness and inclusion problem of general CAs are require transformation to the NFAs.

Our algorithm for testing language inclusion of MCAs can be used in a new approach for testing language inclusion of eREs—we transform eREs to MCAs and apply our algorithm. We implemented our algorithm for testing the language inclusion of MCAs and used Microsoft's Automata library [3], which provides algorithms for transformation of eREs to MCAs and the determinization algorithm of MCAs. We evaluate this approach on eREs from a wide range of applications (e.g., Snort rules [20] used for finding attacks in network traffic) and compared it with the implementation of method when the eREs are transformed to the NFAs [1]. Briefly, the experiments show that the method based on MCAs is less prone the explode. And for the eREs that contain counting operators with large bounds, the method based on MCAs outperforms the method based on NFAs.

The rest of this thesis is organized as follows. Chapter 2 introduces the basics of automata theory (including the definition of a CA), notation used throughout the thesis, and necessary notions from graph theory. In Chapter 3, we present efficient algorithms implementing various operations on NFAs and SFAs. Namely, the algorithm for computing the intersection, computing simulation, and solving the inclusion problem of NFAs and SFAs. In Chapter 4, we introduce several subclasses of CAs and for each of them we give a solution to the emptiness problem. In Chapter 5, we solve the inclusion problem of MCAs. In Chapter 6, we experimentally evaluate the performance of our implementation of the algorithms for testing inclusion problem of MCAs. Chapter 7 summarizes the achieved results and gives the possible further directions of this thesis.

Chapter 2

Automata theory

In this introductory chapter, we introduce all necessary definitions that will be used in the following chapters. First, we define classical finite automata and symbolic finite automata, which we use mainly in Chapter 3 (Section 2.1). Second, we introduce an automata model called *counting automata* (CAs), based on the definition in [14] (Section 2.2). This type of automaton is tha main object of examination in this thesis. Lastly, the values of transitions in the automata are sometimes not important, because we are interested only in the structure of the automata (e.g., whether there exists a path from one state to another state). For such tasks, the automata can be transformed into *directed graphs* (we call them simply *graphs*). Hence in Section 2.3, we introduce basic notions from graph theory.

Throughout the thesis, we use the following notation. We use $\mathbb{N}$ to denote the set of all nonnegative integers $\{0, 1, 2, \dots\}$. The set of all positive integers $\mathbb{N}^+$ is defined as $\mathbb{N} \setminus \{0\}$. The set of the first $n > 0$ positive integers is denoted by $[n] = \{1, 2, \dots, n\}$ and $[0] = \emptyset$. The expression $A \uplus B$ stands for a union of two disjoint sets A, B. Moreover, we extend the notation to use more than two disjoint sets as follows: $\biguplus_{i\in[1]} A_i = A_1$ and $\biguplus_{i\in[n]} A_i = \biguplus_{i\in[n-1]} A_i \uplus A_n$, for $n \geq 2$. Given a function $f : A \to B$, we refer to the elements of f using $a \mapsto b$ (when $f(a) = b$).

2.1 Finite Automata and Symbolic Finite Automata

In the following, suppose that $n \in \mathbb{N}$. A finite, non-empty set Σ of *symbols* is called an *alphabet*. A *string* is a sequence of symbols $a_1a_2 \dots a_n$ where $a_i \in \Sigma$, for $1 \leq i \leq n$. The *length* of w is defined as $|w| = n$. We use $\epsilon \notin \Sigma$ to denote the *empty string*, so $|\epsilon| = 0$. The set of all strings over the alphabet Σ is denoted by Σ^*.

Definition 2.1. A *nondeterministic finite automaton* (NFA) N is a five-tuple $(Q, \Sigma, I, F, \Delta)$ where Q is a finite set of *states*, Σ is an alphabet, $I \subseteq Q$ is the set of *initial states*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*.

Let $N = (Q, \Sigma, I, F, \Delta)$ be an NFA. We use $q \text{-}\{a\}\!\!\to r$ to denote that $(q, a, r) \in \Delta$. A *run* of the NFA N over a string $w = a_1a_2 \dots a_n \in \Sigma^*$ from a state $q_0 \in Q$ is a sequence of transitions $q_0 \text{-}\{a_1\}\!\!\to q_1, q_1 \text{-}\{a_2\}\!\!\to q_2, \cdots, q_{n-1} \text{-}\{a_n\}\!\!\to q_n$. The run is *initial* if $q_0 \in I$, and the run is *accepting* if $q_n \in F$. The string w is *accepted* by N from q if there is some accepting run of N on w from q, otherwise w is *rejected* by N from q. The *language of a state* q is denoted by $\mathcal{L}(N)(q) = \{w \in \Sigma^* \mid w \text{ is accepted by } N \text{ from } q\}$. For convenience, a set of states $P \subseteq Q$ is called a *macro-state*. The definition of the language of a state is lifted to the macro-state R as $\mathcal{L}(N)(R) = \bigcup_{r\in R} \mathcal{L}(N)(r)$. Then the *language of automaton* N is defined as $\mathcal{L}(N) =$

$\mathcal{L}(N)(I)$. The post-image of a state p is defined as $Post(p) = \{p' \mid \exists a \in \Sigma : (p, a, p') \in \Delta\}$ and the post-image of a macro-state P is defined as $Post(P) = \{P' \mid \exists a \in \Sigma : P' = \{p' \mid \exists p \in P : (p, a, p') \in \Delta\}\}$. A *deterministic finite automaton* (DFA) $N = (Q, \Sigma, I, F, \Delta)$ is an NFA where the transition relation Δ is a (partial) function from $Q \times \Sigma$ to Q.

Next, we define symbolic finite automata (SFAs). Informally, an SFA is an NFA, where the transitions are labelled by predicates that denote a set of symbols instead of a single symbol. SFAs can be defined in several ways, for example in [15] simply as an extension of NFAs, where the transition relation Δ is defined as a subset of $Q \times 2^{\Sigma} \times Q$. We use the more complex definition that allows us to have potentially an infinite alphabet (i.e., an alphabet with an infinite number of symbols), following [13]. First, we need to define a notion of an effective Boolean algebra.

Definition 2.2. An *effective Boolean algebra* $\mathcal{A}$ is a six-tuple $(\mathcal{D}, \Psi, [\![\cdot]\!], \wedge, \vee, \neg)$ where Ψ is a set of *predicates* closed under predicate transformers $\vee, \wedge : \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$. A first order interpretation (*denotation*) $[\![\cdot]\!] : \Psi \to 2^{\mathcal{D}}$ assigns to every predicate of Ψ a subset of the *domain* $\mathcal{D}$ such that, for all $\varphi, \psi \in \Psi$ it holds that $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathcal{D} \setminus [\![\varphi]\!]$.

For $\varphi \in \Psi$, we say that φ is *satisfiable* if $[\![\varphi]\!] \neq \emptyset$. The predicate $IsSat(\varphi)$ returns `TRUE` iff φ is satisfiable. The predicate $IsSat$ and the predicate transformers $\wedge, \vee$, and $\neg$ must be effectively computable. We assume that Ψ contains the predicates $\top$ and $\bot$ with $[\![\top]\!] = \mathcal{D}$ and $[\![\bot]\!] = \emptyset$. Let $\Phi \subseteq \Psi$, the set $Minterms(\Phi)$ of *minterms* of a finite set Φ of predicates is defined as the set of all satisfiable predicates of $\{\bigwedge_{\varphi \in \Phi'} \varphi \wedge \bigwedge_{\varphi \in \Phi \setminus \Phi'} \neg\varphi \mid \Phi' \subseteq \Phi\}$ (e.g., see [10] for an algorithm computing minterms).

The symbols from the alphabet of regular expressions are usually encoded in UTF-16 or ASCII, so every symbol can be represented by 16-bit or 8-bit vector. In Examples 2.1 and 2.2, we provide two effective Boolean algebras that implement the operations $\wedge, \vee$, and $\neg$ in a different way.

Example 2.1. For $k > 0$, the $\mathbf{BDD}_k$ algebra is an effective Boolean algebra whose domain $\mathcal{D}$ is the set of all k-bit vectors and predicates Ψ are represented by binary decision diagrams (BDDs) [8] over k Boolean variables $x_1, \ldots, x_k$, representing particular bits of the k-bit vector. The operations $\wedge, \vee$, and $\neg$ directly correspond to the operations on the BDDs. We note that it is necessary to choose the right order of the variables because different order of variables leads to a different BDD, which are different from each other by the size (the number of nodes). Thus the operations above have different time complexity for a different BDD representing the same predicate. The denotation $[\![\cdot]\!]$ of a BDD $\beta \in \Psi$ is the set of k-bit vectors whose binary representation corresponds to a solution of β.

Example 2.2. For $k > 0$, the $\mathbf{BV}_k$ algebra is an effective Boolean algebra whose domain $\mathcal{D}$ is the set of all k-bit vectors and predicates Ψ in bit-vector arithmetic with one free variable x. The operations $\wedge, \vee$, and $\neg$ correspond directly to the standard logical operations on binary vectors. Moreover, the bit-vector arithmetic provides the standard arithmetic operations such as $\leq, <, \geq, >$, and $=$. We use $n_1 \leq x \leq n_2$ as shorthand for $n_1 \leq x \wedge x \leq n_2$. The denotation $[\![\cdot]\!]$ of $\varphi \in \Psi$ is the set of all variables y that makes φ true if x is substituted by y in φ. For example, the regular expression `[a-zA-Z0-9]` can be written as `'0'` $\leq x \leq$ `'9'` $\vee$ `'A'` $\leq x \leq$ `'Z'` $\vee$ `'a'` $\leq x \leq$ `'z'` in $\mathbf{BV}_8$, or more exactly $48 \leq x \leq 57 \vee 65 \leq x \leq 90 \vee 97 \leq x \leq 122$ (in ASCII).

In particular, the algebras $\mathbf{BDD}_8$ or $\mathbf{BV}_8$ represent the encodings ASCII and the algebras $\mathbf{BDD}_{16}$ or $\mathbf{BV}_{16}$ represent the encodings UTF-16. The algebra $\mathbf{BV}_k$ does not

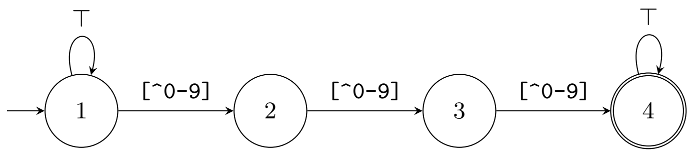


Figure 2.1: An SFA $M = (\{1,2,3,4\}, \mathbf{BV}_8, \{1\}, \{4\}, \{(1,\top,1), (1,\psi,2), (2,\psi,3), (3,\psi,4), (4,\top,4)\})$ where $\psi = (x < 40 \vee 49 > x)$. M denotes the same language as the regular expression `.*[^0-9]{3}.*`.

scale as well as $\mathbf{BDD}_k$ for increasing k (cf. [15]). But the predicates in $\mathbf{BV}_k$ are easy to write as is shown in Example 2.2. Thus we use $\mathbf{BV}_k$ algebra in all examples in this section. Now we are ready to define an SFA.

Definition 2.3. A *symbolic finite automaton* (SFA) M is a five-tuple $(Q, \mathcal{A}, I, F, \Delta)$ where Q is a finite set of *states*, $\mathcal{A} = (\mathcal{D}, \Psi, [\![\cdot]\!], \wedge, \vee, \neg)$ is an effective Boolean algebra, $I \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is a set of *final states*, and $\Delta \subseteq Q \times \Psi \times Q$ is a finite *symbolic transition relation.*

Let $M = (Q, \mathcal{A}, I, F, \Delta)$ be an SFA. Similarly as for NFAs, we use $q\text{-}\{\psi\}\!\!\rightarrow r$ to denote that $(q, \psi, r) \in \Delta$. We write $[\![q\text{-}\{\psi\}\!\!\rightarrow r]\!]$ to denote the set $\{q\text{-}\{a\}\!\!\rightarrow r \mid a \in [\![\psi]\!]\}$ of *concrete transitions* represented by $q\text{-}\{\psi\}\!\!\rightarrow r$. Moreover, let $[\![\Delta]\!] = \bigcup_{q\text{-}\{\psi\}\!\rightarrow r \in \Delta} [\![q\text{-}\{\psi\}\!\!\rightarrow r]\!]$. Other notations (run, language, etc.) are defined analogously as for NFAs. In Figure 2.1 is an example of an SFA.

Example 2.3. Suppose that we want to implement an automaton model that accepts all strings containing a substring of length 3 that does not contain any digit 0–9. In other words, we want to design an automaton M such that $\mathcal{L}(M) = \mathcal{L}(\texttt{.*[\^{}0-9]\{3\}.*})$, where the language of the regular expression is defined as usual. We have two options—design M as an SFA or as an NFA. The SFA M is depicted in Figure 2.1. If M is designed as an NFA, then M would have the same number of states, but each transition ψ of M would be replaced by the set of concrete transitions $[\![\psi]\!]$. Note that the size of $[\![\psi]\!]$ depends on the used encodings (if it uses UTF-16 encoding, then, for example, the self-loop of state 1 is replaced by 2^{16} concrete self-loops).

Definition 2.4. Let $M = (Q, \mathcal{A}, I, F, \Delta)$ be an SFA where $\mathcal{A} = (\mathcal{D}, \Psi, [\![\cdot]\!], \wedge, \vee, \neg)$ is an effective Boolean algebra. We say that M is *complete* if for every state $q \in Q$ and every symbol $a \in \mathcal{D}$, there exists a state r such that $q\text{-}\{\psi\}\!\!\rightarrow r \in \Delta$ with $a \in [\![\psi]\!]$.

SFAs can be completed in this way: we add a new non-final state q_{sink} and from every state $q \in Q$, we add a transition from q to q_{sink} labelled with $\neg \bigvee \{\varphi \mid \exists r \in Q : q\text{-}\{\varphi\}\!\!\rightarrow r \in \Delta\}$, if the disjunction is satisfiable.

2.2 Counting Automata

In this section, we introduce the notion of *counting automata* (CAs), following [14]. Since CAs are defined as a specialisation of a more general model, which is called labelled transition system (LTS), we first define LTS and, next, we extend it to a CA. At the end of this section, we introduced special types of CAs and in Example 2.4 we give a connection between SFAs and CAs.

2.2.1 Labelled Transition Systems

Often, a labelled transition system (LTS) is defined as a triple (Q, A, Δ) where Q is a set of states, A is a set of labels (or actions), and $\Delta \subseteq Q \times A \times Q$ is the transition relation (e.g., [12]). Sometimes, we use the so-called rooted LTS, which is a pair (T, q_0) where $T = (Q, A, \Delta)$ is an LTS and $q_0 \in Q$ is the initial state.

For our purpose the definition of a (rooted) LTS is not sufficient. We want to have some state to be final, i.e., we want to know that some sequence of actions leads to a final state. For more generality, finals and initials state are encoded by formulae. Furthermore, also transition relation is encoded by a formula. We use the following definition [14].

Given a set of variables V and a set of constants Q (disjoint with $\mathbb{N}$), we define a *Q-formula over V* to be a quantifier-free formula φ of Presburger arithmetic extended with constants from Q and Σ, i.e., a Boolean combination of (in-)equalities $t_1 = t_2$ or $t_1 \leq t_2$ where t_1 and t_2 are constructed using $+, \mathbb{N}$, and V, and predicates of the form $x = a$ or $x = q$ for $x \in V, a \in \Sigma$, and $q \in Q$. An assignment M to free variables of φ is a *model* of φ, denoted as $M \models \varphi$, if it makes φ true. The *semantics* of a formula φ is the set $[\![\varphi]\!]$ of all possible tuples of the free variables in φ which make φ true. If $[\![\varphi]\!] \neq \emptyset$, then we say that φ is *satisfiable*. Finally, the predicate $IsSat(\varphi)$ returns `TRUE` iff φ is satisfiable.

Definition 2.5. A *labelled transition system* (LTS) over Σ is a five-tuple $T = (Q, V, I, F, \Delta)$ where

- Q is a finite set of *control states*,
- V is a finite set of *configuration variables*,
- I is the *initial Q-formula* over V,
- F is the *final Q-formula* over V, and
- Δ is the *transition Q-formula* over $V \cup V' \cup \{\mathtt{l}\}$ with $V' = \{x' \mid x \in V\}, V \cap V' = \emptyset$, and $\mathtt{l} \notin V$.

We call $\mathtt{l}$ the symbol variable and allow it as the only term that can occur with a predicate $\mathtt{l} = a$ for $a \in \Sigma$, called an *atomic symbol guard*. Moreover, $\mathtt{l}$ is also not allowed to occur in any other predicates in Δ.

A *configuration* of an LTS T is a function $\alpha : V \to \mathbb{N} \cup Q$ that maps every configuration variable to a number from $\mathbb{N}$ or a state from Q. We will denote by $\mathcal{C}$ the set of all configuration of the LTS T. As mentioned above, the transition relation $[\![\Delta]\!] \subseteq \mathcal{C} \times \Sigma \times \mathcal{C}$ is encoded by the transition formula Δ as follows $(\alpha, a, \alpha') \in [\![\Delta]\!]$ iff $\alpha \cup \{x' \mapsto k \mid \alpha'(x) = k\} \cup \{\mathtt{l} \mapsto a\} \models \Delta$. For a string $w \in \Sigma^*$, we define inductively that a configuration α' is a *w-successor* of α, written $\alpha \xrightarrow{w} \alpha'$, such that $\alpha \xrightarrow{\epsilon} \alpha$ for all $\alpha \in \mathcal{C}$, and $\alpha \xrightarrow{av}$ iff $\alpha \xrightarrow{a} \overline{\alpha} \xrightarrow{v} \alpha'$ for some configuration $\overline{\alpha}$, $a \in \Sigma$, and $v \in \Sigma$. A configuration α is initial if $\alpha \models I$, and final if $\alpha \models F$. The outcome of T on a word w is the set $out_T(w)$ of all w-successors of the initial configurations, and w is accepted by T if $out_T(w)$ contains a final configuration. The *language* $\mathcal{L}(T)$ of T is the set of all words that T accepts.

2.2.2 Definition of Counting Automata

The following definition of counting automaton is a slight modification of the definition in [14].

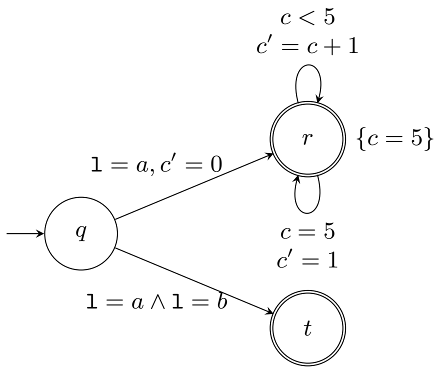


Figure 2.2: An example of a CA $N = (\{q, r, t\}, \{c\}, I, F, \Delta)$ where $F : (\mathtt{s} = r \wedge c = 5) \vee \mathtt{s} = t$, $I : \mathtt{s} = q$, and $\Delta : q\text{-}\{\mathtt{l}{=}a,\top,c'{=}0\}\text{→}r \vee q\text{-}\{\mathtt{l}{=}a\wedge\mathtt{l}{=}b,\top,\top\}\text{→}t \vee r\text{-}\{\top,c{<}5,c'{=}c{+}1\}\text{→}r \vee r\text{-}\{\top,c{=}5,c'{=}1\}\text{→}r$ with $\mathcal{L}(N) = \{w \in \Sigma^* \mid w = az$ where $a \in \Sigma, z \in \Sigma^*$ and $|z| = 5k$ for $k \in \mathbb{N}^+\}$.

Definition 2.6. A *(nondeterministic) counting automaton* (CA) is a five-tuple $A = (Q, C, I, F, \Delta)$ such that (Q, V, I, F, Δ) is an LTS with the following properties:

1. The set of configuration variables $V = C \cup \{\mathtt{s}\}$ consists of a set of counters C and a single control state variable $\mathtt{s}$ such that $\mathtt{s} \notin C$.
2. The transition formula Δ is a disjunction of *transitions*, which are conjunctions of the form $(\mathtt{s} = q) \wedge \sigma \wedge g \wedge f \wedge (\mathtt{s}' = r)$, denoted by $q\text{-}\{\sigma,g,f\}\text{→}r$, where $q, r \in Q$, q is the transition's *guard formula* over $\{\mathtt{l}\}$, g is the transition's *guard formula* over V, and f is the transition's *counter assignment formula*, a conjunction of atomic assignments to counters in which every counter is assigned at most once.
3. There is a constant $\boldsymbol{max}_A \in \mathbb{N}$ such that no counter can ever grow above that value.

Moreover, for every transition $\varphi = q\text{-}\{\sigma,g,f\}\text{→}r$ in Δ, we define the following functions that return particular components of φ: $sym(\varphi) := \sigma$, $cons(\varphi) := g$, and $up(\varphi) := f$. An example of CAs is on Figure 2.2.

Definition 2.7. A *deterministic counting automaton* (DCA) is a CA $N = (Q, C, I, F, \Delta)$ where I has at most one model and, for every symbol $a \in \Sigma$, every reachable configuration α has at most one a-successor.

Example 2.4. We show how to extend CAs to handle large or infinite set of symbols using effective Boolean algebra. We use the idea of the definition of SFAs. A *(nondeterministic) symbolic counting automaton* (SCA) is a six-tuple $N = (Q, \mathcal{A}, C, I, F, \Delta)$, where Q, C, I, F have the same meaning as in Definition 2.6, $\mathcal{A} = (\mathcal{D}, \Psi, [\![\cdot]\!], \wedge, \vee, \neg)$ is an effective Boolean algebra, and Δ is a disjunction of the transitions $(\mathtt{s} = q) \wedge \sigma \wedge g \wedge f \wedge (\mathtt{s}' = r)$ where all components have the same meaning as in Definition 2.6 except that $\sigma \in \Psi$. Similarly as for SFAs, we write $[\![q\text{-}\{\sigma,g,f\}\text{→}r]\!]$ to denote the set of concrete transitions $\{q\text{-}\{\mathtt{l}{=}a,g,f\}\text{→}r \mid a \in [\![\sigma]\!]\}$, and so on. From this example, we see that SCAs are an extension of SFAs. In other words, if $C = \emptyset$ in this definition, then we obtain the definition of SFAs.

SFAs were introduced to reduce the number of transitions in NFAs—if there are multiple transitions between states q and r, then all of them can be replaced by a single transition

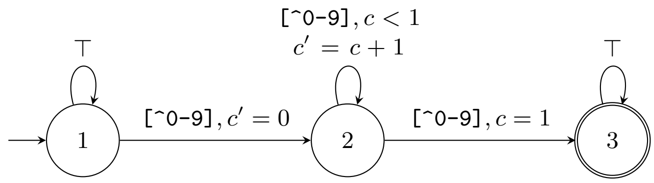


Figure 2.3: An SCA $N = (\{1,2,3\}, \mathbf{BV}_8, \{c\}, I, F, \Delta)$ where $I : \mathtt{s} = 1, F : \mathtt{s} = 3$, and $\Delta : 1 \xrightarrow{\{\top,\top,\top\}} 1 \vee 1 \xrightarrow{\{\psi,\top,c'=0\}} 2 \vee 2 \xrightarrow{\{\psi,c<1,c'=0\}} 2 \vee 2 \xrightarrow{\{\psi,c=1,\psi\}} 3 \vee 3 \xrightarrow{\{\top,\top,\top\}} 3$ with $\psi = (x < 40 \vee 49 < x)$. N denotes the same language as the extended regular expression `.*[^0-9]{3}.*`.

from q to r. For example, the number of transitions remains the same if the alphabet of the regular expression is ASCII or UTF-16 in Figure 2.1. Similarly, we can think that CAs were introduced to reduce the number of states in NFAs. Note that CAs also reduce the number of transitions, but not in as efficient way as SFAs because the symbol guards of transitions can be only a disjunction of $\mathtt{l} = a$ or $\mathtt{l} \neq a$ for $a \in \Sigma$.

Combining the SFAs and CAs as in Example 2.4, we obtain a solid reduction in both number of states and transitions. In Figure 2.3 is an SCA equivalent to the SFA in Figure 2.1. Now suppose that we want to design an SFA and a SCA for the regular expression `.*[^0-9]{k}.*` for $\mathtt{k} > 1$. Note that the SCA for such a regular expression has the same structure as the SCA in Figure 2.3 except that number 1 in the counter guards are replaced by $\mathtt{k} - 2$. On the other hand, the number of states in SFAs for the same regular expression grows linearly with $\mathtt{k}$, thus also the number of transitions. In practice, we have regular expressions where the number of repetitions is larger (e.g., the value of $\mathtt{k}$ in the last example). Finally, we note that all discussions in this thesis about CAs are also true for SCAs or can be easily modified for SCAs.

Lastly, let N be a CA. We often talk about whether a transition is satisfiable or reachable (in N) and whether a final state is reachable (in N) with its satisfiable final condition. We give here the precise definitions of this notation.

Definition 2.8. Let $N = (Q, C, I, F, \Delta)$ be a CA and α any configuration of N.

- We say that a transition $\varphi \in \Delta$ is *reachable* from α if there exists a string $w \in \Sigma^*$ and a configuration β such that β is a w-successor of α and $IsSat(\beta \wedge \varphi)$. Otherwise, φ is *unreachable.*

- We say that a transition $\varphi \in \Delta$ is *satisfiable* from α if $IsSat(\alpha \wedge \varphi)$. Otherwise, φ is *unsatisfiable.*

- We say that a state $q \in Q$ is *reachable* from α if there exists a string $w \in \Sigma^*$ and a configuration β such that β is a w-successor of α and $IsSat(\beta \wedge \mathtt{s} = q)$. Otherwise, q is *unreachable.*

- We say that a state q is a *reachable final state (with its satisfiable final condition φ)* from α if there exists a string $w \in \Sigma^*$ and a configuration β such that β is a w-successor of α with $\beta \models F$ and $IsSat(\beta \wedge \mathtt{s} = q \wedge \varphi)$. Otherwise, q is an *unreachable final state* or its *final condition is unsatisfiable.*

If α is not specified, then it is either clear from the context or it is an initial configuration of N.

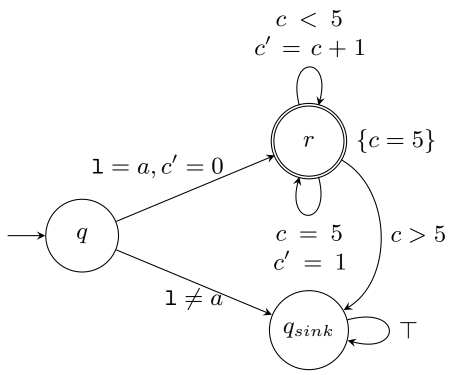


Figure 2.4: An example of a clean and complete CA N.

2.2.3 Types of CAs

Let $N = (Q, C, I, F, \Delta)$ be a CA. We define several special types of CAs that simplify reasoning about them in Chapters 4 and 5. Moreover, for any type we give a procedure that transform any CAs to such type.

Definition 2.9. N is *clean* if for each transition $\varphi \in \Delta$ it holds that $sym(\varphi)$ is satisfiable.

Let $\varphi \in \Delta$, if $sym(\varphi)$ is unsatisfiable, then the whole transition is unreachable, i.e., φ is logically equivalent to $\bot$. So, we can remove φ from N and the language of N still remains the same. We can repeat this process until N contains only reachable transitions.

We note that any transition φ in a clean CA can be still unreachable because we ignore the counter guard of φ. For example, the transition with the counter guard $c > 5 \wedge c < 5$ is unreachable since it is equivalent to $\bot$. In this simple example, it is easy to find that, but it can be difficult in general. Another example is in Figure 2.4, the transition $r \xrightarrow{\{\top, c>5, \top\}} q_{sink}$ is unreachable because there is no way how the counter c can reach to value 6 or more.

Any algorithm for CAs based on finding reachable states builds upon an effective procedure that decides which transitions are reachable. Namely, in Chapters 4 and 5, we find such effective procedures for restricted classes of CAs. It should be mentioned that there is always a possibility to transform every CA into an NFA by unfolding every possible configuration in the CA into part of states in the NFA.

Definition 2.10. N is *complete* if for each configuration α of N and every symbol $a \in \Sigma$ there exists a configuration α' of N such that α' is an a-successor of α.

To make N complete, we first add a new non-final state q_{sink} and the transition $q_{sink} \xrightarrow{\{\top, \top, \top\}} q_{sink}$. For every state q, let $P_q = \{\sigma \wedge g \mid q \xrightarrow{\{\sigma, g, f\}} r \in \Delta^D\}$. Then for every state $q \neq q_{sink} \in Q$, we add a new transitions of $q \xrightarrow{\{\psi\}} q_{sink}$ where $\psi = \wedge_{\varphi \in P_q} \neg\varphi$. Intuitively, if no outgoing transition from q can be executed, then we can use this new added one. For this reason the procedure also preserves determinism.

Example 2.5. In Figure 2.4 is an example of a clean and complete CA. Note that this CA is equivalent to the CA in Figure 2.2, i.e., both CAs recognize the same language.

The following type of automata has an important property—if N is clean, then it preserves the emptiness of language (see Lemma 2.1)—which we use in Chapter 4. To compute such type of CA is easily done directly from the definition.

Definition 2.11. Let $N = (Q, C, I, F, \Delta)$ be a CA. Then the CA $N^T = (Q, C, I, F, \Delta^T)$ is called the *truthfulness of* A where $\Delta^T = \{q \text{-}\{\top,g,f\}\text{→} r \mid q\text{-}\{\sigma,g,f\}\text{→} r \in \Delta\}$.

Lemma 2.1. *Let* $N = (Q, C, I, F, \Delta)$ *be a clean CA. Then* $\mathcal{L}(N) \neq \emptyset$ *if and only if* $\mathcal{L}(N^T) \neq \emptyset$ *where* $N^T = (Q, C, I, F, \Delta^T)$ *is the truthfulness of* A.

Proof. Let α be an initial configuration of N and N^T, i.e., $\alpha \models I$ and $\alpha \models I^T$. First, suppose that $\mathcal{L}(N) \neq \emptyset$. It follows that, there exists a string $w \in \Sigma^*$ such that a final configuration α' is a w-successor of α in N. Since for any formula φ the following holds $[\![sym(\varphi)]\!] \subseteq [\![\top]\!]$, we can conclude that α' is also a w-successor of α in N^T. Thus $\mathcal{L}(N^T) \neq \emptyset$.

Conversely, suppose that $\mathcal{L}(N^T) \neq \emptyset$. Then there is a string $w \in \Sigma^*$ such that a final configuration α' is a w-successor of α in N^T. Note that for any other string z of same length as w, i.e., $|w| = |z|$, α' is a z-successor of α in N^T, because $[\![\top]\!] = \Sigma$. Since N is clean, for every transition φ in N we have $[\![sym(\varphi)]\!] \neq \emptyset$. Thus there must be a string z of the same length as w such that α' is a z-successor of α in N. Thus $\mathcal{L}(N) \neq \emptyset$.

□

Definition 2.12. Let $N = (Q, C, I, F, \Delta)$ be a CA. We say that N is *normalized* if $q\text{-}\{\sigma_1,g_1,f_1\}\text{→} r, q\text{-}\{\sigma_2,g_2,f_2\}\text{→} r \in \Delta$ and $[\![g_1]\!] = [\![g_2]\!]$, $[\![f_1]\!] = [\![f_2]\!]$, then $[\![\sigma_1]\!] = [\![\sigma_2]\!]$.

Every CA N can be normalized by the following procedure: if such two transitions occur in N, then we replace them by the transition $q\text{-}\{\sigma_1 \vee \sigma_2,g_1,f_1\}\text{→} r$ or $q\text{-}\{\sigma_1 \vee \sigma_2,g_2,f_2\}\text{→} r$. It is not hard to see that the language of the automaton is preserved.

2.3 Basic Notions from Graph Theory

A *(finite directed) graph* G is a pair (V, E), where V is a finite set of *vertices* (or *states* if G originates from an automaton) and $E \subseteq V \times V$ is a finite set of *edges* (or *transitions*). If G is a graph, then $V(G)$ and $E(G)$ denote the vertex set of G and the edge set of G, respectively. Let $G = (V, E)$ be a graph. The the *order* (or *size*) of G is $|G|$ defined as $|V|$. The *out-degree* of the vertex $v \in V(G)$ is the number of v' such that $(v, v') \in E$. We say that a graph G' is a *subgraph* of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. G' is called a subgraph of graph $G = (V, E)$ *induced* by a set of vertices $V' \subseteq V$ if $G' = \big(V', E \cap (V' \times V')\big)$.

Graphs G and H are called *isomorphic*, written $G \cong H$, if there exists a bijection $f : V(G) \to V(H)$ such that $(x, y) \in E(G) \iff (f(x), f(y)) \in E(H)$. The graph $C_n = (V, E)$ for some $n \geq 2$ is called the *cycle of length* n (or simply a *cycle* if the length is not important) if $V = [n]$ and $E = \{(i, i + 1) \mid i \in [n - 1]\} \cup \{(n, 1)\}$. We say that a graph G contains a cycle if there is a subgraph of G that is isomorphic to C_n for some $n \geq 2$.

A *path* in G from v_0 to v_n is a sequence $(v_0, v_1, \dots, v_n)$ for $n \geq 0$, where $v_i \neq v_j$ for $i \neq j$ and $(x_{i-1}, x_i) \in E(G)$ for $1 \leq i \leq n$. G is called a *connected* graph if for any two vertices $x, y \in V(G)$ there is a path from x to y, or vice versa. A graph G is a *tree* if G is connected and there is no cycle in G. Let $G = (V, E)$ be a graph. The graph $G - v$, for $v \in V$, denotes an subgraph of G induced by $V - \{v\}$.

Let $N = (Q, C, I, F, \Delta)$ be a CA. The direction of N, written *direction*(N), is a graph $G = (Q, E)$, where $E = \{(q, r) \mid q\text{-}\{\alpha\}\text{→} r \in \Delta\}$. Intuitively, *direction*(N) originates from N if we remove labels from its transitions and ignore that some states are final or initial. The *self-loop* of a CA and a graph G is a transition $q\text{-}\{\alpha\}\text{→} q$ and an edge $(q, q) \in E(G)$, respectively. Note that the self loop in a graph is not considered a cycle.

Chapter 3

An Overview of Efficient Algorithms for FAs and SFAs

In general, for almost every problem there are several algorithms, which differ by simplicity and efficiency (the most simple algorithms are usually not the most efficient ones, and vice versa). The same holds for algorithms for automata, for example the minimization of a DFA (see [16] for two algorithms performing the operation, one running in the time $O(n^2)$ and the other in $O(n \cdot log(n))$, where n is the number of states in the DFA). Unfortunately, there are many problems for which there are no known algorithms running in a time better than exponential in the worst case (e.g., NFA to DFA conversion, which is called *determinization* of NFA, or the inclusion problem of NFAs). In this chapter, we give a brief overview of selected algorithms for NFAs and SFAs. We do not consider DFAs, because the algorithms for this class are easier than for NFAs, see for example [11, Section 4.1] for such algorithms. On top of that, every DFA is also an NFA, thus all algorithms introduced here for NFAs also apply for DFAs.

In Section 3.1, we introduce an algorithm for the intersection of NFAs (SFAs). In Section 3.2, we define the simulation relation on NFAs (SFAs). Moreover, we show how it can be efficiently computed. Simulation is frequently used for accelerating some algorithms for NFAs (SFAs). One example is demonstrated in Section 3.3, namely the inclusion problem of NFAs. Before we start introducing the algorithms, we clarify the meaning of the operations on automata mentioned above plus some other needed later.

Definition 3.1. Let us fix an alphabet Σ. Let N_1, N_2 be automata (classical, symbolic, or counting) with languages $\mathcal{L}(N_1)$ and $\mathcal{L}(N_2)$, respectively. Then,

- the *union* of N_1 and N_2 is the automaton $N_1 \cup N_2$ with $\mathcal{L}(N_1 \cup N_2) = \mathcal{L}(N_1) \cup \mathcal{L}(N_2)$.
- the *intersection* of N_1 and N_2 is the automaton $N_1 \cap N_2$, with $\mathcal{L}(N_1 \cap N_2) = \mathcal{L}(N_1) \cap \mathcal{L}(N_2)$.
- the *set difference* of N_1 and N_2 is the automaton $N_1 \setminus N_2$ with $\mathcal{L}(N_1 \setminus N_2) = \mathcal{L}(N_1) \setminus \mathcal{L}(N_2)$.
- the *complement* of N_1 is the automaton $\overline{N_1}$ with $\mathcal{L}(\overline{N_1}) = \Sigma^* \setminus \mathcal{L}(N_1)$.
- the *universality problem* of N_1 is the problem of deciding whether $L(N_1) = \Sigma^*$.
- the *emptiness problem* of N_1 is the problem of deciding whether $L(N_1) = \emptyset$.
- the *language inclusion problem* of N_1 and N_2 is the problem if deciding whether $L(N_1) \subseteq L(N_2)$.

3.1 Intersection of Two Automata

Initially, we give an algorithm for computation of intersection of NFAs [11, Section 4.2]. Then, we show how to modify the algorithm for computation of intersection of SFAs.

3.1.1 Nondeterministic Finite Automata

Let N_1 and N_2 be NFAs, there are at least two ways how to build the automaton that recognizes the language $\mathcal{L}(N_1) \cap \mathcal{L}(N_2)$. The first one, which is introduced here as Algorithm 1, following [11, Section 4.2], is based on combining runs of both N_1 and N_2. We denote this resulting automaton as $N_1 \cap N_2$. The second one, which is included as part of the solution in Section 3.3, is also based on combining runs, but now of N_1 and N_2', where N_2' is a DFA equivalent to N_2. The size of $N_1 \cap N_2$ is smaller than $N_1 \times N_2$. Since both automata are still NFAs, it is more efficient to use the automaton $N_1 \cap N_2$ for matching or searching. On the other hand, if we want to solve the inclusion problem of N_1 and N_2, then it is preferable to use the automaton $N_1 \times N_2$ (see Section 3.3).

Algorithm 1: Intersection of NFAs

Input : NFAs $N_1 = (Q_1, \Sigma, I_1, F_1, \Delta_1), N_2 = (Q_2, \Sigma, I_2, F_2, \Delta_2)$
Output : NFA $N_1 \cap N_2 = (Q, \Sigma, I, F, \Delta)$ with $\mathcal{L}(N_1 \cap N_2) = \mathcal{L}(N_1) \cap \mathcal{L}(N_2)$

1 $Q, \Delta, F \leftarrow \emptyset$; $I \leftarrow I_1 \times I_2$;
2 $W \leftarrow I$;
3 **while** $W \neq \emptyset$ **do**
4   take and remove (q_1, q_2) from W;
5     $Q \leftarrow Q \cup \{(q_1, q_2)\}$;
6   **if** $q_1 \in F_1$ **and** $q_2 \in F_2$ **then**
7         $F \leftarrow F \cup \{(q_1, q_2)\}$;
8   **foreach** $a \in \Sigma$ **do**
9   **foreach** $q_1' \in \Delta_1(q_1, a), q_2' \in \Delta_2(q_2, a)$ **do**
10   **if** $(q_1', q_2') \notin Q$ **then**
11                 $W \leftarrow W \cup \{(q_1', q_2')\}$;
12             $\Delta \leftarrow \Delta \cup \{((q_1, q_2), a, (q_1', q_2'))\}$;
13 **return** $(Q, \Sigma, I, F, \Delta)$;

For any $q \in Q$ and $a \in \Sigma$, we define $\Delta(q, a) = \{q' \mid (q, a, q') \in \Delta\}$. Algorithm 1 builds the automaton $N_1 \cap N_2$ as follows: the initial set I is the combination of all possible initial states in N_1 and N_2 (Line 1). We use this set to initialize the work list W (Line 2). In the main loop (line 3), until W is empty, we take and remove one pair $p = (q_1, q_2)$ from W, the pair p is a new state of $N_1 \cap N_2$ (Lines 4, 5). On Line 6, we check whether both states in p are final (Line 7), if so then also p is also final (in this step we ensure that strings must be accepted by both N_1 and N_2). Next, we compute the next pairs of states (Lines 8, 9) as combinations of successor of q_1 and q_2 in N_1 and N_2 and glue them together, obtaining the pair $p' = (q_1', q_2')$. Line 10 ensures termination of the algorithm (if some pair is in Q, then the pair was in W, thus it is useless to put it again to W). On Line 12, we add the transition $(q_1, q_2) \xrightarrow{\{a\}} (q_1', q_2')$ to Δ.

3.1.2 Symbolic Finite Automata

Recall that the difference between NFAs and SFAs is that the annotation of the transitions in NFAs are single symbols in contrast to SFAs where the annotation of the transitions are predicates that denote a set of symbols. Thus the algorithm for computation the intersection of two SFAs works similarly as for NFAs. The changes are on Lines 8–13 in Algorithm 1 (and the input/output of the algorithm are SFAs instead of NFAs), which are substituted by the pseudo-code in Algorithm 2. The set $\Delta(q)$ for $q \in Q$ is defined as $\{(q', \alpha) \mid (q, \alpha, q') \in \Delta\}$.

Algorithm 2: Modification of Algorithm 1 (Lines 8–13) for intersection of SFAs

8 **foreach** $(q_1', \alpha) \in \Delta_1(q_1), (q_2', \beta) \in \Delta_2(q_2)$ **do**
9   **if not** $IsSat(\alpha \wedge \beta)$ **then**
10   **continue**;
11   **if** $(q_1', q_2') \notin Q$ **then**
12         $W \leftarrow W \cup \{(q_1', q_2')\}$;
13     $\Delta \leftarrow \Delta \cup \big\{\big((q_1, q_2), \alpha \wedge \beta, (q_1', q_2')\big)\big\}$;
14 **return** $(Q, \Sigma, I, F, \Delta)$;

Let $p = (q_1, q_2)$ be a pair taken from W (Line 4 in Algorithm 1). For each $(q_1', \alpha) \in \Delta_1(q_1)$ and $(q_2', \beta) \in \Delta_2(q_2)$, we check the satisfiability of $\alpha \wedge \beta$ (Line 9). If $\alpha \wedge \beta$ is unsatisfiable, then there is no way to go to q_1' from q_1 via α and to q_2' from q_2 via β, simultaneously. Otherwise, there are such symbols for which the previous is true. The set of these symbols is are denoted by the label $\alpha \wedge \beta$ (Line 13). Line 11 has the same meaning as Line 10 in Algorithm 1. For yet another method of computation intersection of SFAs, we refer the reader to [15].

3.2 Simulation Relation

First, we define a notion of the simulation relation on NFAs and SFAs, respectively. Second, we demonstrate an algorithm for computing simulation on NFAs [13]. The demonstrated algorithm, called INY, is a slightly modified version of the algorithm from [18]. Finally, we show two algorithms for the computation of a simulation on SFAs [13], called GlobINY and NoCount.

In this section, we use the following notation. Let $A_1, \dots, A_n$ be sets. If $R \subseteq A_1 \times \cdots \times A_n$ is an n-ary *relation*, for $n \geq 2$, then $R(x_1, \dots, x_{n-1}) := \{y \in A_n \mid R(x_1, \dots, x_{n-1}, y)\}$ for any $x_1 \in A_1, \dots, x_{n-1} \in A_{n-1}$. If $n = 2$ and $A = A_1 = A_2$, R is called a *binary relation on A*.

Definition 3.2. Let $N = (Q, \Sigma, I, F, \Delta)$ be an NFA. A binary relation S on Q is a *simulation* on N if whenever $(q, r) \in S$, then the following conditions hold:

(i) if $q \in F$, then $r \in F$, and

(ii) for all $a \in \Sigma$ and $q' \in Q$ such that $q \xrightarrow{\{a\}} q' \in \Delta$, there is a state r' such that $r \xrightarrow{\{a\}} r' \in \Delta$ and $(q', r') \in S$.

Definition 3.3. Let $M = (Q, \mathcal{A}, I, F, \Delta)$ be an SFA. A binary relation S on Q is a *simulation* on M if whenever $(q, r) \in S$, then the following two conditions hold:

(i) if $q \in F$, then $r \in F$, and

(ii) for all $a \in \mathcal{D}$ and $q' \in Q$ such that $q \xrightarrow{a} q' \in [\![\Delta]\!]$, there is a state r' such that $r \xrightarrow{a} r' \in [\![\Delta]\!]$ and $(q', r') \in S$.

There exists a unique maximal simulation relation[1] on N, which is reflexive and transitive. Such a unique maximal simulation relation on N is called *simulation preorder* on N. Similar remarks also hold for SFAs. In fact, all demonstrated algorithms below compute the simulation preorder on NFAs or SFAs.

3.2.1 Nondeterministic Finite Automata

In the following, we describe the INY algorithm [18], given as Algorithm 3. We use a slightly modified version from [13].

Algorithm 3: INY

Input : An NFA $N = (Q, \Sigma, I, \Delta, F)$
Output : The simulation preorder $\preceq_N$

```
1  for p, q ∈ Q, a ∈ Σ do
2  |  N_a(q, p) ← |Δ(q, a)|;
3  Sim ← Q × Q;
4  NotSim ← F × (Q \ F) ∪ {(i, j) | ∃a ∈ Σ : Δ(i, a) ≠ ∅ ∧ Δ(j, a) = ∅};
5  while NotSim ≠ ∅ do
6  |  remove some (i, j) from NotSim and Sim;
7  |  for t -{a}-> j ∈ Δ do
8  |  |  N_a(t, i) ← N_a(t, i) − 1;
9  |  |  if N_a(t, i) = 0 then
10 |  |  |  for s -{a}-> i ∈ Δ such that (s, t) ∈ Sim do
11 |  |  |  |  NotSim ← NotSim ∪ {(s, t)};
12 return Sim;
```

On Lines 1 and 2 we initialize all counters $N_a(q, p)$, individually for every triple (p, q, a) where $p, q \in Q$ and $a \in \Sigma$. The value of the counter denotes the number of states r' satisfying the condition (ii) of Definition 3.2; at the start the value overapproximates the real value. Initially, *Sim* stores all pairs of states, but at the end of the algorithm *Sim* contains the simulation preorder (Line 3). The set *NotSim* stores all pairs (i, j) in which we are sure that i is not simulated by j. In the beginning, *NotSim* contains all pairs (i, j) such that $i \in F$ and $j \notin F$, because such pairs do not satisfy the condition (i) in Definition 3.2. Since N is not complete, we need to also add to *NotSim* all pairs (i, j) for which the condition (ii) in Definition 3.2 is not trivially satisfied. That is, the pair (i, j) is added to *NotSim* if there is at least one symbol a for which we can go from i via a to some other state, but from j there is no outgoing transition via a (Line 4). Until *NotSim* is not empty, we remove the pair (i, j) from *Sim* and *NotSim* (Line 5). By the definition of *NotSim*, we know that i is not simulated by j. Thus for all states t and symbols a such that $t \xrightarrow{a} j \in \Delta$, we know that the state j do not satisfies the condition (ii) of Definition 3.2, because i is not simulated by j. So, the counter $N_a(t, i)$ decreases (Lines 7, 8). If the counter $N_a(t, i)$ is zero

[1] The simulation relation S on an NFA N is maximal, if when there is another simulation relation S' on N, then we have $S' \subseteq S$

(Line 9), then we know that there is no states t' such that $t \xrightarrow{a} t' \in \Delta$ and i is simulated by t'. Thus we add all pairs (s,t) to $NotSim$ if $s \xrightarrow{a} i \in \Delta$ because we know that s is not simulated by t, since we can go from s to i via a and there is no transition $t \xrightarrow{a} t' \in \Delta$ such that i is simulated by t'.

3.2.2 Symbolic Finite Automata

An SFA M is *globally mintermised* if the set $\Psi_\Delta = \{\varphi \mid \exists q, r : q \xrightarrow{\varphi} r \in \Delta\}$ of the predicates appearing on its transitions forms a partition on $\bigcup_{\varphi \in \Delta} [\![\varphi]\!]$. Every SFA can be made globally mintermised (by process called *global mintermisation*) by replacing each $q \xrightarrow{\varphi} r \in \Delta$ with the set of transitions $\{q \xrightarrow{\omega} r \mid \omega \in Minterms(\Psi_\Delta) \text{ and } IsSat(\omega \wedge \varphi)\}$ (see [10] for an efficient algorithm).

Let M be a globally mintermised SFA. Then M has the following property: for any predicate $\alpha, \beta \in \Psi$, if $\alpha \neq \beta$ then $[\![\alpha]\!] \cap [\![\beta]\!] = \emptyset$. Hence we can look at the labels in the transitions of M as syntactic elements and apply Algorithm 3. The whole process is demonstrated in Algorithm 4, called GlobINY [13]. Note that in this way any algorithm for NFAs can be used also for SFAs.

Algorithm 4: GlobINY

Input : An SFA $M = (Q, \mathcal{A}, I, \Delta, F)$
Output : The simulation preorder $\preceq_N$

$\Delta_G \leftarrow$ globally mintermised Δ;
return INY$((Q, \Psi_{\Delta_G}, I, \Delta, F))$;

This approach is valid as shown in [13], but not the most efficient one. The problem is that the number of minterms of the set Φ is in the worst case $2^{|\Phi|}$. There exists a modification of GlobINY using only local mintermisation the so-called algorithm LocalMin [13] (M is said to be *local mintermised* if for every state $q \in Q$, the set $\Psi_{\Delta,q} = \{\varphi \in \Psi \mid \exists r : q \xrightarrow{\varphi} r \in \Delta\}$ of the predicates used on the transition starting from q forms a partition). The advantage of local mintermisation over the global is that the number of transitions grows only exponentially to the maximum number of outgoing transitions of a state. Both methods GlobINY and LocalMin for computing simulation preorder on SFAs are based on counting, which requires that the SFAs are at least local mintermised. We introduce one more algorithm, called NoCount, which is not based on counting. Experimental results in [13] show that NoCount overall outperformed LocalMin and GlobINY.

Let $M = (Q, \mathcal{A}, I, \Delta, F)$ be an SFA. Let us define the formula φ_{si} for $s, i \in Q$ to denote $\bigvee_{(s,\psi,i) \in \Delta} \psi$. For a given set $S \subseteq Q$ and a state $q \in Q$, we use $\Gamma(q, S)$ to denote the disjunction of all predicates that reach S from q, i.e., $\Gamma(q, S) = \bigvee_{s \in S} \varphi_{tj}$. We also write $q \to S$ to denote that there is a transition $q \xrightarrow{\psi} s \in \Delta$ such that $s \in S$.

Algorithm 5, called NoCount, works as follows. Similarly as in INY, initially, Sim stores all pairs of states and $NotSim$ stores all pairs (i,j) such that i is not simulated by j (Lines 1, 2). Since M is complete, the initial values of $NotSim$ consists of only pairs (i,j) that do not satisfy the condition (i) in Definition 3.3. Until $NotSim$ is empty (Line 3), we proceed in the following way. By the definition of the set $NotSim$ we know that each state $j \in NotSim(i)$ does not simulate i, hence all these states are removed from $Sim(i)$ (Line 5). The information of $NotSim(i)$ was processed, so we set $NotSim(i)$ to $\emptyset$ (Line 6). For each state $t \in Rm$, which is defined on Line 4, we initialize the formula ψ as the disjunction of all predicates from t to the states that are simulated by i (Lines 7, 8). For each state s

Algorithm 5: NoCount

Input : A complete SFA $M = (Q, \mathcal{A}, I, \Delta, F)$
Output : The simulation preorder $\preceq_M$

1 $\mathit{Sim} \leftarrow Q \times Q$;
2 $\mathit{NotSim} \leftarrow F \times (Q \setminus F)$;
3 **while** $\exists i \in Q : \mathit{NotSim}(i) \neq \emptyset$ **do**
4     $\mathit{Rm} \leftarrow \{t \mid t \rightarrow \mathit{NotSim}(i)\}$;
5     $\mathit{Sim}(i) \leftarrow \mathit{Sim}(i) \setminus \mathit{NotSim}(i)$;
6     $\mathit{NotSim}(i) \leftarrow \emptyset$;
7   **for** $t \in \mathit{Rm}$ **do**
8         $\psi \leftarrow \Gamma(t, \mathit{Sim}(i))$;
9   **for** $s \xrightarrow{\varphi_{si}} i \in \Delta$ such that $(s,t) \in \mathit{Sim}$ **do**
10   **if** $\mathit{IsSat}(\neg\psi \wedge \varphi_{si})$ **then**
11                 $\mathit{NotSim} \leftarrow \mathit{NotSim} \cup \{(s,t)\}$;
12 **return** Sim;

such that $(s,t) \in \mathit{Sim}$ and $s \xrightarrow{\varphi_{si}} i \in \Delta$ (Line 9), we ask whether there is a symbol a such that we can make move from s to i and we cannot make move from t to any state that is simulated by i (Line 10). If so, then the condition (ii) of Definition 3.3 is not satisfied. Thus we add (s,t) to NotSim (Line 11).

The reason for introducing Rm on Line 4 is optimization. We could remove a single pair (i,j) from NotSim and Sim and go to Line 7, similarly as in INY, but this is inefficient. To see that, let $j, j' \in \mathit{NotSim}(i)$ and suppose there is a transition from t to both j and j'. Then Lines 7–11 are independent of whether the pair (i,j) or (i,j') is taken from NotSim; that is, the formula ψ and φ_{si} are exactly the same in both iterations of (i,j) and (i,j'). Thus it is only important whether there is some transition from t to some state in $\mathit{NotSim}(i)$, i.e., $t \rightarrow \mathit{NotSim}(i)$.

3.3 Inclusion Problem of NFAs

Let N_1 and N_2 be NFAs. Recall that the inclusion problem of N_1 and N_2 is the problem of deciding whether $\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2)$. Note that

$$\begin{aligned}\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2) &\Leftrightarrow \mathcal{L}(N_1) \setminus \mathcal{L}(N_2) = \emptyset \\ &\Leftrightarrow \mathcal{L}(N_1) \cap \overline{\mathcal{L}(N_2)} = \emptyset \\ &\Leftrightarrow \mathcal{L}(N_1) \cap \mathcal{L}(\overline{N_2}) = \emptyset.\end{aligned}$$

The classical algorithm is based on the last equivalence—it builds the so-called product automaton $N_1 \times \overline{N_2}$ of N_1 and the complement of N_2 and checks whether the language of the product automaton is empty, i.e., searches for a final state. Since we do not need the whole language of the product automaton, $\mathcal{L}(N_1 \times \overline{N_2}) = \mathcal{L}(N_1) \cap \mathcal{L}(\overline{N_2})$, but we need to only know whether such a language is empty, it is not necessary to build the whole product automaton and then search for a final state. It can all be done on-the-fly using the fact that if we encounter a final state, then the algorithm can stop, because we find a string w such that $w \in \mathcal{L}(N_1) \cap \mathcal{L}(\overline{N_2})$, thus we find that $\mathcal{L}(N_1) \not\subseteq \mathcal{L}(N_2)$. For an example of such an algorithm see [11, Section 4.2].

Nevertheless, the inclusion problem of NFAs is PSPACE-complete, there are optimized algorithms for the inclusion problem of NFAs that outperform the classical algorithm in many cases (cf. [7]). These optimized algorithms using the simulation to prune out unnecessary search path in the search for a final state. In this section, we demonstrate the optimized algorithm from [7]. Before we give the optimized algorithm, we first introduce terminology from [7, Sections 3,5].

Let $N = (Q, \Sigma, I, F, \Delta)$ be an NFA, recall that a set of states in N is called a macro-state (see also Section 2.1 for other related definitions). A macro-state R is *accepting* if it contains at least one state r such that $r \in F$, otherwise R is *rejecting*. For two macro-states P and R, we write $P \preceq^{\forall\exists} R$ as a shorthand for $\forall p \in P. \exists r \in R : p \preceq r$. We use $N^{\subseteq}$ to denote the set of relations over the states of N that imply language inclusion. Lemma 3.1 shows that any simulation relation $\preceq$ on N is in $N^{\subseteq}$.

Lemma 3.1. *Given a simulation $\preceq$ on an NFA N, $q \preceq r \implies \mathcal{L}(N)(q) \subseteq \mathcal{L}(N)(r)$.*

Let $N_1 = (Q_1, \Sigma, I_1, F_1, \Delta_1)$ and $N_2 = (Q_2, \Sigma, I_2, F_2, \Delta_2)$ be NFAs. A state in the *product automaton* $N_1 \times \overline{N_2}$ is a pair (p, P) where p is a state in N_1 and P is a macro-state in N_2, such a pair (p, P) is called a *product-state*. A product-state is *accepting* if p is an accepting state in N_1 and P is a rejecting macro-state in N_2. The language of N_1 is not contained in the language of N_2 iff there exists some accepting product state (p, P) reachable from some initial product-state. We use $\mathcal{L}(N_1 \times \overline{N_2})(p, P)$ to denote the language of the product-state (p, P) in $N_1 \times \overline{N_2}$. Note that $\mathcal{L}(N_1 \times \overline{N_2})(p, P) = \mathcal{L}(N_1)(p) \setminus \mathcal{L}(N_2)(P)$. The union of N_1 and N_2 is the automaton $N_1 \cup N_2 = (Q_1 \uplus Q_2, \Sigma, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$. Lemma 3.2 provides resources for the optimizations of the classical algorithm.

Lemma 3.2. *Let N_1, N_2 be NFAs, $(p, P), (r, R)$ be two product-states, where p, r are states in N_1 and P, R are macro-states in N_2, and $\preceq$ be a relation in $(N_1 \cup N_2)^{\subseteq}$. Then, $p \preceq r$ and $R \preceq^{\forall\exists} P$ implies $\mathcal{L}(N_1, N_2)(p, P) \subseteq \mathcal{L}(N_1, N_2)(r, R)$.*

The first optimization, referred to as Optimization 1, is based on the following. Suppose that we encounter a product-state (p, P) in the process of building the product automaton. Assume that a product-state (r, R) was already encountered. If we would know that $\mathcal{L}(N_1 \times \overline{N_2})(p, P) \subseteq \mathcal{L}(N_1 \times \overline{N_2})(r, R)$, then we can stop searching from (p, P) because every string that takes (p, P) to an accepting product-state will also take (r, R) to an accepting product-state. But it is difficult to decide whether $\mathcal{L}(N_1 \times \overline{N_2})(p, P) \subseteq \mathcal{L}(N_1 \times \overline{N_2})(r, R)$ before the whole product-automaton is built. For this purpose we can use Lemma 3.2—we stop searching from the product-state (p, P), if we already encountered a product-state (r, R) with $p \preceq r$ and $R \preceq^{\forall\exists} P$. The simulation can be computed in polynomial time, but it is incomplete—simulation implies language inclusion, but not vice versa. Nevertheless, we obtain only partial information about state language inclusion using simulation, the results in [7] show that this approach is more efficient than the classical algorithm.

Optimization 2 is based on the observation that $\mathcal{L}(N_1, N_2)(p, P) = \emptyset$ if there is a state $p' \in P$ such that $p \preceq p'$. It follows from that $\mathcal{L}(N_1)(p) \subseteq \mathcal{L}(N_2)(P)$ and so $\mathcal{L}(N_1)(p) \setminus \mathcal{L}(N_2)(P) = \emptyset$, if there is a state $p' \in P$ such that $p \preceq p'$. Thus if we encounter a product-state (p, P) with such property, we do not continue generate other successors of (p, P), because they are all rejecting product-states.

Moreover, let $p_1, p_2 \in P$. Note that $(p, P) \preceq (p, P \setminus \{p_1\})$ if $p_1 \preceq p_2$. It follows from Lemma 3.2, since $P \preceq^{\forall\exists} P \setminus \{p_1\}$ and $P \setminus \{p_1\} \preceq^{\forall\exists} P$. Thus every product-state (p, P) can be reduced to (p, P') such that there is no $p_1, p_2 \in P'$ with $p_1 \preceq p_2$ or $p_2 \preceq p_1$. If the product-state has such a property, then we say that it is in the *minimal form*. For any product state

Algorithm 6: Language inclusion checking

Input : NFAs $N_1 = (Q_1, \Sigma, I_1, F_1, \Delta_1), N_2 = (Q_2, \Sigma, I_2, F_2, \Delta_2)$, and a relation $\preceq \in (N_1 \cup N_2)^{\subseteq}$

Output : `TRUE` if and only if $\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2)$.

1 **if** there is an accepting product state in $\{(i, I_2) \mid i \in I_1\}$ **then**
2   **return** `FALSE`;
3 $\mathit{Processed} \leftarrow \emptyset$;
4 $\mathit{Worklist} \leftarrow \mathit{Initialize}(\{(i, \mathit{Minimize}(I_2)) \mid i \in I_1\})$
5 **while** $\mathit{Worklist} \neq \emptyset$ **do**
6   Pick and remove a product-state (r, R) from $\mathit{Worklist}$;
7     $\mathit{Processed} \leftarrow \mathit{Processed} \cup \{(r, R)\}$;
8   **foreach** $(p, P) \in \{(r', \mathit{Minimize}(R')) \mid (r', R') \in \mathit{Post}((r, R))\}$ **do**
9   **if** (p, P) is an accepting product-state **then**
10   **return** `FALSE`;
11   **if** $\nexists p' \in P$ such that $p \preceq p'$ **then**
12   **if** $\nexists (s, S) \in \mathit{Processed} \cup \mathit{Worklist}$ s.t. $p \preceq s \wedge S \preceq^{\forall\exists} P$ **then**
13   Remove all (s, S) from $\mathit{Worklist} \cup \mathit{Processed}$ s.t. $s \preceq p \wedge P \preceq^{\forall\exists} S$;
14                 $\mathit{Worklist} \leftarrow \mathit{Worklist} \cup \{(p, P)\}$;
15 **return** `TRUE`;

(p, P), we write $(p, \mathit{Minimize}(P))$ to denote its minimal form. Using minimization we can prune out some unnecessary search path, because $\mathit{Post}((p, \mathit{minimize}(P))) \subseteq \mathit{Post}((p, P))$ where $\mathit{Post}((p, P))$ is the post-image of the product-state (p, P) defined as $\mathit{Post}((p, P)) = \{(p', P') \mid \exists a \in \Sigma : (p, a, p') \in \Delta_1, P' = \{p'' \mid \exists p \in P : (p, a, p'') \in \Delta_2\}\}$.

The classical algorithm augmented by the optimizations above is given as Algorithm 6, following [7, Section 5]. If some initial product-state is accepting, then the language inclusion of N_1 and N_2 does not hold (Lines 1, 2). In the set *Processed*, we store all visited product-states (Line 3). The algorithm starts searching from the initial product-states, but these initial product-states are first reduced to their minimal forms (Line 4). Until $\mathit{Worklist}$ is empty, we pick and remove a product-state (r, R) from $\mathit{Worklist}$ (Line 6). The product-state (r, R) is also moved to *Processed* (Line 7). On Line 8 we generate all successors (p, P) of (r, R), but, again, we reduce them to their minimal forms. If (p, P) is accepting, then the language inclusion does not hold of N_1 and N_2 (Lines 9, 10). In the standard algorithm, the product state is always added to $\mathit{Worklist}$, unless the product-state is not in *Proccessed*. In the optimized version, we first ensure that there is no $p' \in P$ such that $p \preceq p'$, otherwise we can stop searching from (p, P) by Optimization 2 (Line 11). Second, we ensure that there is no product-state (s, S) in *Processed* or $\mathit{Worklist}$ such that $s \preceq p \wedge P \preceq^{\forall\exists} S$, otherwise we can stop searching from (p, P) by Optimization 1 (Line 12). If the previous two conditions are satisfied, then we move (p, P) to $\mathit{Worklist}$ and also remove from *Processed* and $\mathit{Worklist}$ all product-states (s, S) such that $s \preceq p$ and $P \preceq^{\forall\exists} S$, because they are useless by Optimization 1 (Lines 13, 14). If no generated product-state is accepting, then we know that $\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2)$, thus we return `TRUE`.

For the correctness of the algorithm see [7, Section 5]. Algorithm 6 can be also used for the universality problem of N_2, if N_1 is one-state NFAs with $\mathcal{L}(N_1) = \Sigma^*$.

Chapter 4

Emptiness problem of CAs

Recall that the emptiness problem of CA N is the problem of deciding whether $\mathcal{L}(N) = \emptyset$ (see Definition 3.1). Since any CA N can be transformed to an NFA N' by unfolding every possible configuration in N into part of states in N', we can use all existing algorithms for NFAs also for CAs. In particular, we can use the algorithm for testing the emptiness of NFAs [11, Section 4.2], we call this solution *trivial*.

To the best of our knowledge, no general solution is known for the emptiness problem of CAs except the trivial one. In this chapter, we are able to solve the emptiness problem without unfolding the CAs to NFAs if the input CAs meet the given conditions. These conditions then define a subclass of CAs. In Section 4.1, we introduce the subclass of CAs from [14], the so-called *monadic counting automata* (MCAs), which naturally arise from extended regular expressions. The solution for this subclass is straightforward, but it serves some important observations, which we use later. Moreover, we show where lies the difficulty in developing an algorithm solving the emptiness problem of general CAs. In Section 4.2, we define a new subclass of CAs—*looping counting automata* (LCAs). This subclass is the base case for the recursive definition of the another subclass of CAs, called *advanced looping counting automata* (ALCAs), which are defined in Section 4.3. In both sections, we provide the algorithm for the emptiness problem of LCAs and ALCAs, respectively. The examples and figures in this chapter demonstrate that LCAs and ALCAs are capable of representing more complex extended regular expressions (but still not beyond the regular languages). Finally, we also conclude that ALCAs are a wider class than LCAs and LCAs are a wider class than MCAs. In symbols, we can roughly write MCAs $\subset$ LCAs $\subset$ ALCAs. For the rest of this chapter, let us fix Σ to be an alphabet.

4.1 Monadic Counting Automata

In this section, we introduce Monadic Counting Automata (MCAs) following the definition in [14, Section 4.1]. Such automata naturally arise from the *extended regular expressions* (eREs). The abstract syntax of eREs is

$$R ::= \emptyset \mid \varepsilon \mid \sigma \mid R_1R_2 \mid R_1 + R_2 \mid R^* \mid \sigma\{m, n\}$$

where σ is a predicate denoting a set of alphabet symbols, and $m, n \in \mathbb{N}$ such that $m \leq n$. The semantics is defined as in the standard regular expressions (REs), with $\sigma\{m, n\}$ denoting a string w with $m \leq |w| \leq n$ symbols each of them satisfying σ.

Note that eREs still denote the class of regular languages, but in a more succinctly way than the standard REs. Thus also MCAs denote the class of regular languages, but in a more succinctly way than NFAs. For convenience, we write the elements from the set of symbols that σ denotes within brackets `[]` unless the set contains only a single symbol. In such case, the brackets are omitted (e.g., instead of the eRE `[a]{0,2}` we write `a{0,2}`). Moreover, if σ is equal to `.`, then σ denotes the whole alphabet Σ. For example, the eRE `[abc]{5,5}` denotes all strings of length 5 where each symbol is `a`, `b`, or `c`.

Definition 4.1. A (nondeterministic) *monadic counting automaton* (MCA) is a CA $M = (Q, C, I, F, \Delta)$ where the following holds:

1. The set of control states $Q = Q_s \uplus Q_c$, where Q_s is a set of *simple states* and Q_c is a set of *counting states*.

2. The set of counters $C = \{c_q \mid q \in Q_c\}$ consists of a unique counter c_q for every counting state $q \in Q_c$.

3. All transitions containing counter guards or updates must be incident with a counting state in the following manner. Every counting state $q \in Q_c$ has a single *increment transition*, a self-loop $q \text{-}\{\sigma, c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}\!\!\rightarrow q$ with the value of c_q limited by the bound $\boldsymbol{max}_q$ of q, and possibly several *entry transitions* of the form $r \text{-}\{\sigma, g, c'_q = 0\}\!\!\rightarrow q$ which set c_q to 0, where g is $\top$ or a counter guard containing only c_r if r is a counting state. As for *exit transitions*, every counting state is either *exact* or *range*, where exact counting states have exit transitions of the form $q \text{-}\{\sigma, c_q = \boldsymbol{max}_q, f\}\!\!\rightarrow s$ and range counting states have exit transitions of the form $q \text{-}\{\sigma, \top, f\}\!\!\rightarrow s$ with $s \in Q$ such that $s \neq q$, where f is $\top$ or $c'_s = 0$ if s is a counting state.

4. The initial condition I is of the form

$$I : \bigvee_{q \in Q_s^I} \mathtt{s} = q \vee \bigvee_{q \in Q_c^I} (\mathtt{s} = q \wedge c_q = 0)$$

   for some sets of initial simple and counting states $Q_s^I \subseteq Q_s$ and $Q_c^I \subseteq Q_c$, respectively.

5. The final condition F is of the form

$$F : \bigvee_{q \in Q_s^F \cup Q_r^F} \mathtt{s} = q \vee \bigvee_{q \in Q_e^F} (\mathtt{s} = q \wedge c_q = \boldsymbol{max}_q)$$

   where $Q_s^F \subseteq Q_s$ is a set of simple final states, $Q_r^F \subseteq Q_r$ is a set of final range counting states, and $Q_e^F \subseteq Q_e$ is a set of final exact counting states.

More precisely, the MCAs arise from the extended regular expressions if the subexpressions $\sigma\{m, n\}$ appear only in the forms of $\sigma\{n, n\}$ or $\sigma\{0, n\}$. This is without loss of generality since $\sigma\{m, n\}$ can be rewritten as $\sigma\{m, m\}\sigma\{0, n - m\}$. Usually we write $\sigma\{n\}$ instead of $\sigma\{n, n\}$ (e.g., the last eRE `[abc]{5,5}` can be rewritten as `[abc]{5}`).

MCAs are a subclass of CAs, hence every MCA can be transformed into a clean CA (see the procedure below Definition 2.9). Recall that this transformation preserves the language of the automaton. Moreover, the transformation also preserves the conditions in Definition 4.1 because after removing any transition from an MCA all conditions in Definition 4.1 are still true. In other words, if we transform any MCA to a clean CA, then the CA is again an MCA. An example of an MCA is in Figure 4.1.

We give a solution to the emptiness problem of MCAs. From the preceding paragraph, we can assume without loss of generality that $M = (Q, C, I, F, \Delta)$ is a clean MCA. Furthermore,

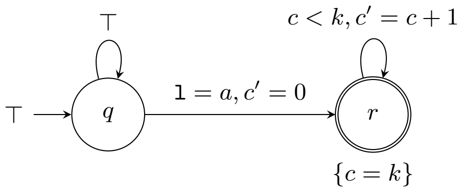


Figure 4.1: Example of an MCA M denoting the same language as the eRE `.*a.{k}`, where $k \in \mathbb{N}$.

we can assume that for any transition $\varphi \in [\![\Delta]\!]$ we have $[\![sym(\varphi)]\!] = \top$, which is justified by Lemma 2.1.

In M, each exit transition φ from a simple state or a range counting state is reachable, because they do not contain counter guards, i.e., $cons(\varphi) = \top$. It remains to check whether the exit transitions from the exact counting states are reachable. The counting state q has one incremental transition of the form $q \text{-}\{\top, c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}\!\!\rightarrow q$ and possibly several exit transitions of the form $q \text{-}\{\top, c_q = \boldsymbol{max}_q, f\}\!\!\rightarrow s$. Thus these transitions are also reachable. Intuitively, we can execute the self-loop of q as long as the condition $c_q = \boldsymbol{max}_q$ is not satisfied, eventually the condition become satisfiable, so also the exit transitions (see Section *Acceleration of Self-loops* for more details).

To find a final state $q \in Q_s^F \cup Q_r^F \cup Q_e^F$ we simply apply some searching algorithm on $direction(M)$ from the initial states in M. If q is found and q is an exact counting state, i.e., $q \in Q_e^F$, then we also need to check whether its final condition is satisfiable. But we know this already—the final condition is satisfiable—since the condition $c_q = \boldsymbol{max}_q$ in the final condition of q is the same as the counter guard in the exit transitions of q. Therefore, if a final state is found, then $\mathcal{L}(M) \neq \emptyset$. Otherwise, $\mathcal{L}(M) = \emptyset$. The time complexity of the algorithm is $\mathcal{O}(n + m)$ where n is the number of states and m is the number of transitions in M.

Acceleration of Self-loops

Let $M = (Q, C, I, F, \Delta)$ be a clean MCA. Suppose that $q \in Q_c$ is a counting state. By the definition of MCAs we know that there is a self-loop of the form $q \text{-}\{\sigma, c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}\!\!\rightarrow q$. Moreover, any entry transition from r to q is of the form $r \text{-}\{\sigma, g, c'_q = 0\}\!\!\rightarrow q$. It is not hard to see that the possible value of c_q can be represented by the formula $0 \leq c_q \leq \boldsymbol{max}_q$. Note that there is no difference if q is an exact or a range counting state. The only difference is that we can leave the state q only if $c_q = \boldsymbol{max}_q$ when q is an exact counting state (and if $c_q \leq \boldsymbol{max}_q$ when q is a range counting state).

The purpose of the preceding paragraph is noting that we do not need executing the self-loop of q one by one to decide whether an outgoing transition (or a final condition) of q is satisfiable reachable. In the case of MCAs, it is trivial since the bounds of the self-loops and the exit transitions of the exact counting states are the same (by the definition). But in the general CAs, there is no hope that the bounds are always the same. Although they are the same, the exit transition can be still unreachable for several reasons.

For example, suppose that q has a self-loop of the form $q \text{-}\{\sigma, c < \boldsymbol{max}_c, c' = c + 2\}\!\!\rightarrow q$ and the exit transition of the form $q \text{-}\{\sigma, c = \boldsymbol{max}_c, f\}\!\!\rightarrow r$ where $\boldsymbol{max}_c \in \mathbb{N}$. It seems that the reachability of the exit transition depends on whether $\boldsymbol{max}_c$ is even or odd number, but this is not true. The exit transition can be reachable for both even and odd value of $\boldsymbol{max}_c$. It depends on whether we enter the state q with even or odd value of c. The value of c can be changed anywhere in the CA. Thus we need to explore the whole CA (consider every configuration of the CA) if no extra information is given. This example demonstrates a difficulty of the emptiness problem of the general CAs. The same situation occurs in any problem of CAs, which is based on deciding whether the outgoing transition of some state is reachable (e.g., the language inclusion of CAs).

The observation from the first paragraph of this section can be generalized. Suppose that the value of c_q is known if we are in a counting state q. Then the value of c_q can be updated by the formula

$$\varphi = \exists k : (0 \leq k \leq \boldsymbol{max}_q \wedge c'_q = c_q + k \wedge c'_q \leq \boldsymbol{max}_q).$$

Using this formula we can easily test whether c_q can obtain a value $n \in \mathbb{N}$ in q by checking $IsSat(\varphi \wedge (c_q)' = n)$. In particular, we can test whether the exit transitions of an exact counting state q are reachable by putting $n = \boldsymbol{max}_q$. Because we do not have to execute the self-loop one by one to obtain the information, we call this formula an *acceleration* formula of the self-loop. Note that the length of the formula is independent on $\boldsymbol{max}_q$.

4.2 Looping Counting Automata

In this section, we introduce looping counting automata (LCAs) in a similar way as MCAs and give a solution to the emptiness problem of LCAs. The LCAs serve the basis of the recursive definition of an advanced looping counting automata (ALCAs), which are introduced in the following section. For the rest of this section let $k_1, \ldots, k_n \in \mathbb{N}$.

Definition 4.2. A *looping counting automaton* (LCA) is a CA $A = (Q, C, I, F, \Delta)$ where the following holds:

1. The set of control states Q is partitioned into non-empty sets $Q^1, Q^2, \ldots, Q^n$ for some $n \geq 1$, i.e., $Q = Q^1 \uplus Q^2 \uplus \cdots \uplus Q^n$.
2. Each block Q^i is further partitioned into a set of simple states Q^i_s, a set of counting states Q^i_c, and a set with the main state Q^i_m such that $|Q^i_m| = 1$. The only state in Q^i_m is called the *main state* (or *interface*) of Q^i_m and is denoted by q^i_*. So, $Q^i = Q^i_s \uplus Q^i_c \uplus Q^i_m$.
3. The set of counters $C = C^1 \uplus C^2 \uplus \cdots \uplus C^n$ where $C^i = \{c_q \mid q \in Q^i_c \uplus Q^i_m\}$ consists of a unique counter c_q for every counting state and the main state in the block Q^i. The unique counter of the main state of Q^i is called the *main counter* of Q^i and is denoted by c^i_*.
4. If there is a transition from $q_i \in Q^i$ to $q_j \in Q^j$ such that $i \neq j$, then $q_i = q^i_*$ and $q_j = q^j_*$, i.e., q_i is the main state of Q^i and q_j is the main state of Q^j. Such transitions are called *outside* transitions. The other transitions are called *inside* transitions.
5. Let $G = (Q, E) = direction(A)$. Let G^i be a subgraph of G induced by Q^i for each $i \in [n]$. Then no graph $G^i - q^i_*$ contains a cycle.
6. For each $i \in [n]$, the main state q^i_* of Q^i has:
   (I) possibly several *entry outside transitions* of the form $q^j_* \text{-}\{\sigma,g,f\}\text{\textrightarrow} q^i_*$ where q^j_* is the main state of Q^j with $i \neq j$ and $f := (c^i_*)' = 0$ and possibly several *entry inside transition* of the form $q \text{-}\{\sigma,g,f\}\text{\textrightarrow} q^i_*$ where $q^i_* \neq q \in Q^i$ and $f := \top$ or $f := \big((c^i_*)' = c^i_* + 1\big)$.

(II) possibly several *exit outside transitions* of the form $q^i_* \xrightarrow{\{\sigma,g,f\}} q^j_*$ where q^j_* is the main state of Q^j with $i \neq j$ and $g := (c^i_* = k_i)$ or $g := \top$ and possibly several *exit inside transitions* of the form $q^i_* \xrightarrow{\{\sigma,g,f\}} q$ where $q^i_* \neq q \in Q^i$, $g := (c^i_* < k_1)$, and $f := \top$ or $f := (c'_q = 0)$ if q is a counting state.

(III) at most one *incremental transition*, a self-loop $q^i_* \xrightarrow{\{\sigma, c^i_* < k_i, (c^i_*)' = c^i_* + 1\}} q^i_*$, or a self-loops of the form $q^i_* \xrightarrow{\{\sigma,\top,\top\}} q^i_*$.

7. For each $i \in [n]$, all transitions containing counter guards or updates must be incident with the main state as described in the point 6 above. or with a counting state in the following manner. Every counting state $q \in Q^i_c$ has:

   (I) possibly several *entry inside transitions* of the form $r \xrightarrow{\{\sigma,g,f\}} q$ where $q \neq r \in Q^i$ and $f := (c'_q = 0)$ or $f := (c'_q = 0 \wedge (c^i_*)' = c^i_* + 1)$.

   (II) possibly several *exit inside transitions* of the form $q \xrightarrow{\{\sigma,g,f\}} r$ where $q \neq r \in Q_i$ and $g := (c_q = \boldsymbol{max}_q)$ if q is an exact counting state or $g := \top$ if q is a range counting state.

   (III) a single *incremental transition*, a self-loop of the form $q \xrightarrow{\{\sigma, c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}} q$ or $q \xrightarrow{\{\sigma, c_q < \boldsymbol{max}_q, c'_q = c_q + 1 \wedge (c^i_*)' = c^i_* + 1\}} q$.

8. The initial condition I is of the form:

$$\bigvee_{q^i_* \in Q^*} \left(\mathsf{s} = q^i_* \wedge (c^i_*)' = 0 \right)$$

   where $Q^* \subseteq \bigcup_{i \in [n]} Q^i_m$.

9. The final condition F is of the form:

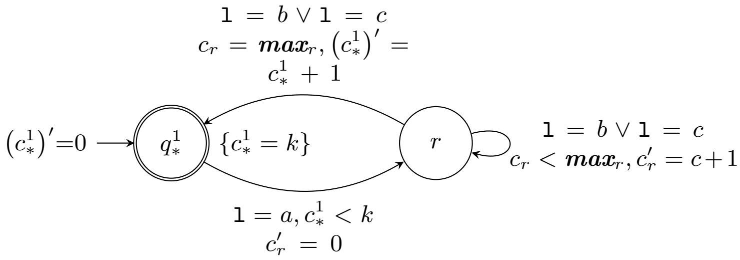
$$\bigvee_{q^i_* \in Q^*_1} \left(\mathsf{s} = q^i_* \wedge c^i_* = k_i \right) \vee \bigvee_{q^i_* \in Q^*_2} \left(\mathsf{s} = q^i_* \right)$$

   where $Q^*_1, Q^*_2 \subseteq \bigcup_{i \in [n]} Q^i_m$ such that $Q^*_1 \cap Q^*_2 = \emptyset$.

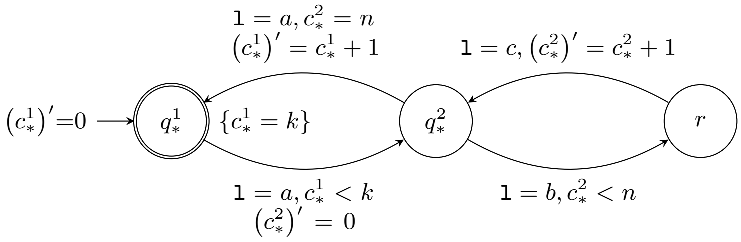
Note that every LCA can be transformed into a clean LCA by the procedure given below Definition 2.9 because after removing any transition from the LCA all conditions in Definition 4.2 are still true. In Figure 4.2 are examples of an LCA and a non-LCA. Another example of an LCA is in Figure 4.3.

We give a brief argument showing that the class of LCAs is wider than the class of MCAs. Let $M = (Q, C, I, F, \Delta)$ be an MCA and suppose that $1 \leq |Q| = n$. The set of states Q can be partitioned into $Q^1, \ldots, Q^n$ such that $|Q^i| = 1$ for $i \in [n]$. Each state $q_i \in Q$ becomes the main state of the block Q^i. The counter c_q of q becomes the main counter of Q^i. Since every main state must have the main counter, we add a new main counter for each simple state. Note that using this construction (after adding some more technical details) all conditions in Definition 4.2 are satisfied. In fact, we show that LCAs are at least as wide as MCAs, the fact that LCAs are wider than MCAs is witnessed by the example in Figure 4.2 (a). Before we solve the emptiness problem of LCAs, we need to define a property of an LCA, which we use in the next section.

Definition 4.3. Let $N = (Q, C, I, F, \Delta)$ be an LCA. We say that N is *single partitioned* if Q can be partitioned into a single block Q^1.



(a) A looping counting automaton



(b) A non-looping counting automaton

Figure 4.2: In (a) is an example of an LCA N_1 accepting the same language as the regular expression `(a[bc]{n}){k}` where $\boldsymbol{max}_r = n-1$ with $n \in \mathbb{N}^+$ and $k \in \mathbb{N}$. In (b) is an example of non-LCA N_2 accepting the same language as the regular expression `(a(bc){n}a){k}` where $n, k \in \mathbb{N}$. The semantics of $(\sigma)\{n\}$ is defined inductively as $(\sigma)\{n\} := \sigma(\sigma)\{n-1\}$ and $(\sigma)\{0\} := \varepsilon$.

4.2.1 Emptiness problem of LCAs

Analogously to the emptiness problem of MCAs, we assume without loss of generality that $N = (Q, C, I, F, \Delta)$ is a clean LCA and for any transition $\varphi \in [\![\Delta]\!]$ we have $[\![sym(\varphi)]\!] = \top$. We begin with several observations: (1) All transitions except the exit transitions from the main states in N are always reachable because these transitions are similar to the transitions in the MCAs. The difference is that the inside transitions in the LCAs can also contain an update on the main counter, but this update does not cause the transitions become unreachable. (2) Only the transitions in a block Q^i can change the value of c_*^i. (3) Only the main states can be initial and final. Observations (2) and (3) follow directly from the definition of LCAs.

To show that $\mathcal{L}(N) \neq \emptyset$ we need to find a reachable final state from the initial states and check whether its final condition is satisfiable. To find a reachable final state we need to decide whether the exit outside transitions from the main states are reachable, because only these transitions lead to the other main states (and only these states can be final). The exit outside transitions from the main states have the counter guard of the form $c_*^i = k_i$ or $\top$. Note that the final condition of the final state is also either $c_*^i = k_i$ or $\top$. Thus checking whether the final condition is satisfiable is the same as checking whether the exit outside transition is reachable. In other words, only one procedure, which decides whether the exit transitions from the main state are reachable, is sufficient.

By the observation (2) only the transitions between the states in a block Q^i can change the value of c_*^i. Thus the reachability of the exit transition from q_*^i depends only on the block Q^i. Suppose that the exit outside transition from c_*^i has the counter guard of the form $c^i_* = k_i$ (if the counter guard is $\top$, then this transition is clearly reachable). We need to develop a procedure $IsSat(Q^i, k_i)$ that takes a block Q^i and the value $k_i \in \mathbb{N}$ and produces a `TRUE` or `FALSE` answer depending on whether it is possible to reach the main state of Q^i such that the value of c^i_* is equal to k_i. For a moment, assume that we have such a procedure.

Since only the main states are initial and final by observation (3), it is sufficient to search in the graph G^* where G^* is a subgraph of $direction(N)$ induced by $\bigcup_{i\in[n]} Q^i_m$. We apply some search algorithm on G^* starting from the initial states with the following modification. Suppose that we are in the state $q^i_* \in V(G^*)$. If there is an edge from q^i_* to q^j_*, then q^j_* is visited only if the exit outside transition from q^i_* to q^j_* in N is reachable. Suppose that this exit transition has the counter guard of the form $c^i_* = k_i$, otherwise this transition is always reachable. Then q^j_* is visited only if $IsSat(Q^i, k_i)$ returns `TRUE`. We note that every state is visited at most once. As proposed before, the same procedure is used for checking whether the final condition of q^i_* is satisfiable. If the final condition is $\top$, then we have immediately $\mathcal{L}(N) \neq \emptyset$. So suppose that the final condition is $c^i_* = k_i$. If $IsSat(Q^i, k_i)$ returns `TRUE`, then $\mathcal{L}(N) \neq \emptyset$, otherwise we need to continue searching. If no final state with its satisfiable final condition is found, then $\mathcal{L}(N) = \emptyset$.

Algorithm 7: Checking whether the main counter may have the value k_i if the main state is reached

Input : A clean LCA $N = (Q, C, I, F, \Delta)$, a partition $Q^i \subseteq Q$, and $k_i \in \mathbb{N}$
Output : `TRUE` if and only if it is possible to reach the main state c^i_* with the value of $c^i_* = k_i$

if $q^i_* \xrightarrow{\{c^i_*<k_i,(c^i_*)'=c^i_*+1\}} q^i_* \in \Delta$ **then**
    return `TRUE`;
$Worklist \leftarrow \{(q^i_*, 0, 0)\}$;
$Processed \leftarrow \emptyset$;
while $Worklist \neq \emptyset$ **do**
    pick and remove $(q, low, high)$ from $Worklist$;
    if q is an exact counting state **then**
        $low \leftarrow low + \boldsymbol{max}_q$;
        $high \leftarrow high + \boldsymbol{max}_q$;
    if q is a range counting state **then**
        $low \leftarrow low$;
        $high \leftarrow high + \boldsymbol{max}_q$;
    foreach $q \xrightarrow{\{\top,g,f\}} r \in \Delta$ such that $r \in Q^i$ **do**
        if f contains the counter update $(c^i_*)' = c^i_* + 1$ **then**
            $low' \leftarrow low + 1$; $high' \leftarrow high + 1$;
        else
            $low' \leftarrow low$; $high' \leftarrow high$;
        if $r = q^i_*$ **then**
            $Processed \leftarrow Processed \cup \{(low', high')\}$;
        else
            $Worklist \leftarrow Worklist \cup \{(r, low', high')\}$;
return $solve(Processed, k_i)$;

From now, if we say that the transition $\varphi \in [\![\Delta]\!]$ is *executed* in N, then we mean that the current configuration α of N is replaced by the a-successor of α where $a \in [\![sym(\varphi)]\!]$. The procedure $IsSat(Q^i, k_i)$ is implemented in Algorithm 7. First, if there is the increment self-loop of q^i_*, then we return `TRUE` because the self-loop of q^i_* has the same structure as the self-loop of a counting state in the MCAs (Lines 1, 2). If $(q, low, high) \in Worklist \subseteq Q^i \times \mathbb{N} \times \mathbb{N}$, then there is a sequence of transitions from q^i_* to q in N such that if we execute these transitions, then the value of c^i_* is increased by k where $low \leq k \leq high$. If $(low, high) \in Processed \subseteq \mathbb{N} \times \mathbb{N}$, then there is a sequence of transitions beginning and ending with q^i_* in N such that if we execute these transitions, then the value of c^i_* is increased by k where $low \leq k \leq high$. Initially on Lines 3 and 4, $Worklist$ stores only a triple $(q^i_*, 0, 0)$, since all entry outside transitions to q^i_* have the update $(c^i_*)' = 0$. And $Processed$ is initialized to the empty set, since we have not found a non-empty sequence starting and ending with q^i_* yet.

The purpose of the main loop is to enumerate sequences starting and ending with q^i_* and remember how the value of c^i_* is updated. In such sequences, every state appears at most once (except the main states) because the self-loops of the states are accelerated. Until $Worklist$ is empty, we take and remove $(q, low, high)$ from $Worklist$ (Lines 5, 6). If q is a counting state, then the incremental self-loop of q can also contains the increment of c^i_* besides the increment of c_q. As shown in Section 4.1, the value of c_q is between 0 and $\boldsymbol{max}_q$ if q is reached (the possible update of c^i_* does not changed this observation). In other words, the value of c_q is increased by k where $0 \leq k \leq \boldsymbol{max}_q$. Thus if the update of the self-loop contains the increment of c^i_*, then also c^i_* is increased by k where $0 \leq k \leq \boldsymbol{max}_q$. Suppose that q is an exact counting state. As shown in Section 4.1, we can leave the state q if $c_q = \boldsymbol{max}_q$. Thus if we want the leave the state q then the c_q must be increased by the value $k = \boldsymbol{max}_q$. Thus also c^i_* is increased by $\boldsymbol{max}_q$ (Lines 7–9). On other hand, if q is a range counting state, then we can leave the state for any value of $c_q \leq \boldsymbol{max}_q$. In other words, if we leave the state q then the value of c_q is increased by k where $0 \leq k \leq \boldsymbol{max}_q$. By the same value of k is increased the value of c^i_* (Lines 10–12).

After we accelerate the self-loop of q, we enumerate all successors $r \in Q^i$ of q (Line 13). If the transition $q\text{-}\{\sigma,g,f\}\!\!\rightarrow r$ contains the update on c^i_*, then the value of c^i_* is increased by one (Lines 14, 15). Otherwise, the value of c^i_* is not changed (Lines 16, 17) If $r = q^i_*$, then we found a sequence of transitions beginning and ending with q^i_*. We also know how the values of c^i_* are changed if these transitions are executed—this information is stored in low and $high$. Thus the pair $(low, high)$ is added to $Processed$ (Lines 18, 19). Otherwise, we add the triple $(r, low, high)$ to $Worklist$ (Lines 20, 21).

If $Worklist$ is empty, then we enumerate all possible sequences (without repetition of inner states in the sequence) of transitions starting and ending with q^i_*. Every such sequence of transitions is stored in $Processed$ as pair $(low, high)$. These sequences can be executed several times in a different order. Thus we need to decide whether there is a sequence of numbers $x_1, \ldots, x_n$ such that $\Sigma_{i \in [n]} x_i = k_i$ and for every x_i there is a pair $(low, high) \in Proccesed$ such that $low \leq x_i \leq high$. The purpose of function $solve(Processed, k_i)$ is to find a sequence of x_i satisfying the property above. If $solve(Processed, k_i)$ found the sequence of x_i, then returns `TRUE`; otherwise returns `FALSE` (Line 20).

Let d be the maximum out-degree of a state in G^i. The number of cycles in G^i is bounded above by $|G^i|^d$. Each cycle corresponds to one sequence starting and ending with q^i_*. The time complexity of Algorithm 7 is $\mathcal{O}(S(|G^i|^d))$ where $S(\cdot)$ is the function representing the time complexity of $solve(Processed, k_i)$, which depends on the number of sequences and the value of k_i.

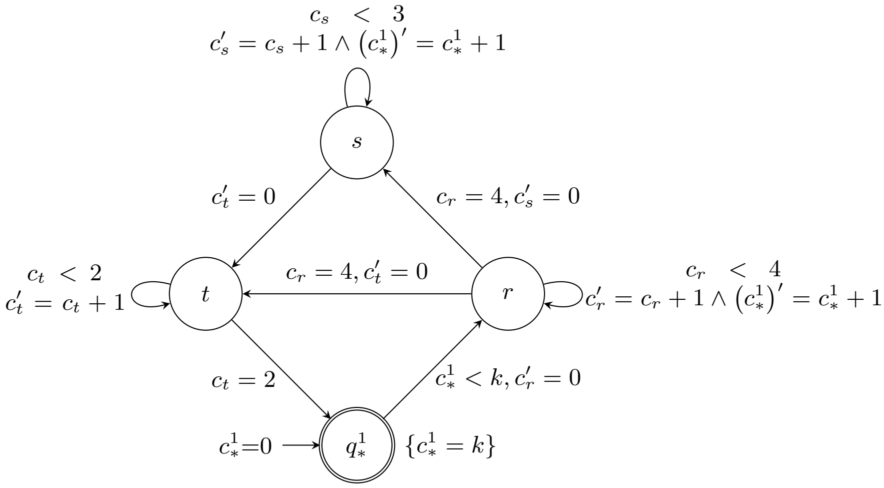


Figure 4.3: An example of LCA N where $k \in \mathbb{N}$.

Example Demonstrating Algorithm 7

We demonstrate the solution to the emptiness problem of the LCA $N = (Q, C, I, F, \Delta)$, which is given in Figure 4.3. Since N has a single partition, Algorithm 7 directly gives the answer to the emptiness problem of N.

We start with $Worklist = \{(q^1_*, 0, 0)\}$ and $Processed = \emptyset$. The number in the parentheses represents how many times we execute the body of the while loop. (1) We remove $(q^1_*, 0, 0)$ from $Worklist$. Since q^1_* is the main state without the self-loop, we process only the exit transitions from q^1_*. There is only one transition from q^1_* to r. This transition does not contain the update on q^1_*. Thus $(r, 0, 0)$ is added to $Worklist$. (2) We remove $(r, 0, 0)$ from $Worklist$ where r is an exact counting state. Note that r has the incremental self-loop containing the update on c^i_* with the bound $\boldsymbol{max}_r = 4$. So the value of c^i_* is increased by 4. We examine all exit transitions from r. No transitions contain the update on c^i_*, hence $Worklist = \{(t, 4, 4), (s, 4, 4)\}$. (3) We remove $(t, 4, 4)$ from $Worklist$. The state t is an exact counting state but does not contain an update on c^i_*, so we only calculate exit transitions from r. Since there is only one outgoing transition to q^i_* without update on c^1_*, we add $(4, 4)$ to $Processed$. (4) In $Worklist$ remains the triple $(s, 4, 4)$. Since s is a range counting state with the self-loop containing the update on c^1_* and the exit transition to t does not contain the update on c^1_* we add $(t, 4, 7)$ to $Worklist$. (5) We remove $(t, 4, 7)$ from $Worklist$. Analogously to (3), the self-loop and the exit transition of t do not change the value of c^1_*, so $(4, 7)$ is added to $Processed$.

After that $Worklist = \emptyset$ and $Processed = \{(4, 4), (4, 7)\}$. For example, let $k = 13$. Then there are several possible solutions, e.g., the sequences $7, 6$ or $4, 4, 5$ are both solutions. Clearly, if $0 < k < 4$, then there is no solution. Moreover, it can be shown that for any $k \geq 4$ there exists a solution.

4.3 Advanced Looping Counting Automata

In this section, we extend the class of LCAs as follows. First, we define a new binary automaton operation. Second, by using this operation we show how to construct a wider

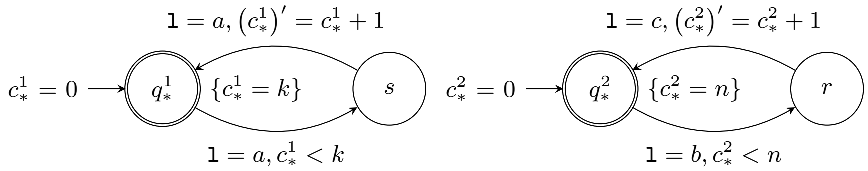


Figure 4.4: Let $k, n \in \mathbb{N}$. On the left is an LCA N_1 and on the right is an LCA N_2. Note that N_1 and N_2 are both single partitioned LCAs and also both are ALCAs.

subclass of CAs than LCAs, which is called *advanced looping counting automaton* (ALCAs). Lastly, we give a solution to the emptiness problem of ALCAs, which is somewhat a repeated application of Algorithm 7.

Definition 4.4. Let $N = (Q_N, C_N, I_N, F_N, \Delta_N)$ and $L = (Q_L, C_L, I_L, F_L, \Delta_L)$ be a CA and a single partitioned LCA, respectively, such that $Q_N \cap Q_L = \{q\}, C_N \cap C_L = \emptyset$, and q is the main state of Q_L. Then the *(automaton) addition* of N and L is the CA $N \oplus L = (Q_N \cup Q_L, C_L \cup C_N, I_N, F_N, \Delta'_N \vee \Delta_L)$ where Δ'_N originates from Δ_N by replacing every disjunct of the form $r \xrightarrow{\{\sigma,g,f\}} q$ by $r \xrightarrow{\{\sigma,g,f\wedge I_L\}} q$ and every disjunct of the form $q \xrightarrow{\{\sigma,g,f\}} r$ by $q \xrightarrow{\{\sigma,g\wedge F_L,f\}} r$ where $q \neq r \in Q_N$.

Definition 4.5. The class of all *advanced looping counting automata* (ALCAs) is defined inductively as follows:

1. Every LCA is an ALCA.
2. Let $N_1 = (Q_1, C_1, I_1, F_1, \Delta_1)$ be an ALCA, and $N_2 = (Q_2, C_2, I_2, F_2, \Delta_2)$ be a single partitioned LCA such that $Q_1 \cap Q_2 = \{q\}$ and $C_1 \cap C_2 = \emptyset$ where q is the main of Q_2. Then $N_1 \oplus N_2$ is an ALCA. Moreover, the single state $q \in Q_1 \cap Q_2$ is called the *bridge* state and the block Q^i with $q \in Q^i$ is extended to $Q^i \cup Q_2$.

The main state and the main counter of Q^i are are still denoted as q^i_* and c^i_*.

We note that only main states in the ALCAs can be final (and initial), since the final (and initial) condition is not changed through the construction. Clearly, the LCA in Figure 4.2 (a) is an ALCA. Moreover, the non-LCA in Figure 4.2 (b) is also an ALCA (see Example 4.1).

Example 4.1. Let N_1 and N_2 be the LCAs in Figure 4.4. Since N_1 is an LCA, it is also an ALCA. Thus $N_1 \oplus N_2$ is defined if the state s in N_1 is renamed to q^2_*. The resulting automaton is depicted in Figure 4.2 (b). The state q is the main state of the block $\{q^1_*, q^2_*, r\}$ in N and the main state q^2_* of $\{q^1_*, r\}$ in N_2 is the bridge state in N of the block $\{q^1_*, q^2_*, r\}$.

4.3.1 Emptiness Problem of ALCAs

We begin with the following lemma, which we use to show that our approach is valid in the end of this section. Lemma 4.1 shows that in some cases the addition operation is commutative. For the rest of the section, let $n, m \in \mathbb{N}^+$.

Lemma 4.1. *Let $N = (Q_N, C_N, I_N, F_N, \Delta_N)$ be an ALCA and $L_1 = (Q_1, C_1, I_1, F_1, \Delta_1)$, $L_2 = (Q_2, C_2, I_2, F_2, \Delta_2)$ be single partitioned LCAs such that $N \oplus L_1$ and $N \oplus L_2$ are both defined. If $Q_1 \cap Q_2 = \emptyset$ and $C_1 \cap C_2 = \emptyset$, then*

$$N \oplus L_1 \oplus L_2 = N \oplus L_2 \oplus L_1.$$

Proof. Assume that $Q_1 \cap Q_2 = \emptyset$ and $C_1 \cap C_2 = \emptyset$. We know that $Q_N \cap Q_2 = \{q\}$ where q is the main state of Q_2, otherwise $N \oplus L_2$ is not defined. Since $Q_1 \cap Q_2 = \emptyset$, we have $(Q_N \cup Q_1) \cap Q_2 = \{q\}$ where q is the main state of Q_2. Moreover, $(C_N \cup C_1) \cap C_2 = \emptyset$. Thus $N \oplus L_1 \oplus L_2$ is defined. It can be also shown that $N \oplus L_2 \oplus L_1$ is defined by interchanging the subscript 1 and 2. The equality follows from the fact that union and disjunction are both commutative operations.

□

Let $N = (Q, C, I, F, \Delta)$ be a clean ALCA and $Q^1, \ldots, Q^n$ be the blocks of Q. Suppose that for any transition $\varphi \in [\![\Delta]\!]$ we have $[\![sym(\varphi)]\!] = \top$. Analogously to the LCAs, only the main states in N can be initial and final. Therefore it is also sufficient to search for a final state in the graph G^* where G^* is a subgraph of $direction(N)$ induced by the set of the main states, i.e., by the set $\bigcup_{i \in [n]} Q^i_m$. We can apply the same procedure as described in Section 4.2.1 but with the different implementation of $IsSat(Q^i, k_i)$ because the blocks are more complex (note that only the transitions in the block Q^i can change the value of c^i_*).

The implementation of $IsSat(Q^i, k_i)$ is straightforward if we know exactly how N is built. Suppose that $L_1, \ldots, L_m$ is the sequence of LCAs such that $N = L_1 \oplus \cdots \oplus L_m$. Note that L_i is a single partitioned LCA for each $2 \leq i \leq m$.

Let $i := m$, we implement $IsSat(Q^i, k_i)$ as follows. Suppose that q is the main state of L_i. We check whether the final condition of L_i is satisfiable by Algorithm 7. If the final condition is not satisfiable (the algorithm returns `FALSE`), then we remove q (and all transitions, which are connected with q) from L_j for each $j < i$, since we know that the exit transitions from q in N are not unreachable. We set $i := n - 1$ and repeat the process until $i = 1$. Finally, if $i = 1$, then L_1 is a general LCA. We use the solution described in Section 4.2.1. Then $\mathcal{L}(N) = \emptyset$ iff $\mathcal{L}(L_1) = \emptyset$.

In general, the process of how N is constructed is not known. Our purpose is to use the procedure described in the preceding paragraph. Thus we need to find a way how to reveal the structure of N. In particular, we develop an algorithm that reveals the structure of a single block Q^i. Then we apply this algorithm on all other blocks in N to reveal the structure of N.

Let Q^i be a block in N. The bridge states of Q^i are known or can be identified as follows. Each bridge state contains the entry transitions with the update $(c_q)' = 0$, the exit transitions with the counter guard $c < k$, and possibly the exit transitions with the counter guard either $c = k$ or $\top$. We note that the same structure of the transitions can have also the main state of Q^i. Thus we need to add the condition that the bridge state must not be the main state.

Let G^i be a subgraph of $direction(N)$ induced by Q^i. We simply apply the depth-first search on G^i starting from the main state of Q^i with the following modifications: (1) We remember the sequences of visited states for each possible path. In other words, we generate a tree representing these paths. Hence the root of the tree is the main state of Q^i. (2) We stop searching in a particular search path if we encounter on the bridge state or the main state for the second time. We note that it is impossible to encounter a state q twice if q is not the main or bridge state (if such state q appears, then N is not an ALCA).

Suppose that we have such a tree T for G^i. Recall that the leaves of T are either the bridge states or the main state. For each leaf ℓ_i we create a set $L_i = \{\ell_i\}$ (note that we can create fewer sets than the number of leaves since some leaves can be the same). For each ℓ_i we proceed from the leaf to the root such that each state q is added to L_i until $q = \ell$. Then

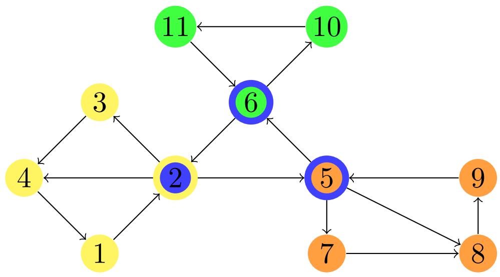


Figure 4.5: The graph $direction(Q^i)$ of a block Q^i of some ALCA with the main state $1 \in Q^i$. The colours of the states indicate how the block Q^i is built where the outer colour denotes the original colour of the state (e.g., we see that the yellow state 2 was replaced by the blue states 2, 5, and 6).

every L_i represents the set of states of a particular LCA. It remains to define the sequence of L_i, in which N was constructed.

We start the sequence S with $L_{q_*^i}$. Suppose that we have already created the sequence S. Then we add to S all L_i such that they are not already in S and have a non-empty intersection with some set, which is already in S (the exact order between these L_i is not important by Lemma 4.1). This algorithm for a single block of N is demonstrated in Example 4.2.

Briefly, we show why this algorithm works. Clearly, every state in Q^i is in some L_i and $L_i \subseteq Q^i$. The path from the leaf q to the next appearance q identifies the cycle in G^i. Thus states appearing in this path must be in the one single partitioned LCA. All paths from the leaf q to the next appearance of q identify all possible cycles in the LCA. Thus if we take all these states in the paths, then we obtain all states specifying one particular LCA. These states are exactly stored in L_j for some j. It should be mentioned that there is a possibility to have the states in a single partitioned LCA from which there is no way to get back to the main state. If such states appear, then these states can be omitted without change of the language of the automaton since these states are not final. The order of the L_i is obvious. Different order of L_i leads to undefined operation of $\oplus$ or it not change anything by the existence of Lemma 4.1.

Using the algorithm, we find the single partitioned LCAs from which Q^i is built, but we also find all sequences starting and ending with the main state of the particular LCA. Thus if we modify the algorithm to remember how the value of the main counter of the LCA is changed, then we can use immediately $solve(Processed, k_i)$ (we suppose that this information is again stored in $Processed$). Then the time complexity of such an algorithm is the same as the time complexity of Algorithm 7 times the number of LCAs from which the ALCAs is built, i.e., the number of the sets L_i.

Example 4.2. Let Q^i be a block of some ALCAs such that the $direction(Q^i)$ is depicted in Figure 4.5. The tree T that is generated from $direction(Q^i)$ by the algorithm described in this section is depicted in Figure 4.6. From T we see that $2, 6$, and 5 are bridge states of Q^i. Moreover, $L_1 = \{1, 2, 3, 4\}, L_2 = \{2, 5, 6\}, L_5 = \{5, 7, 8, 9\}$, and $L_6 = \{6, 10, 11\}$.

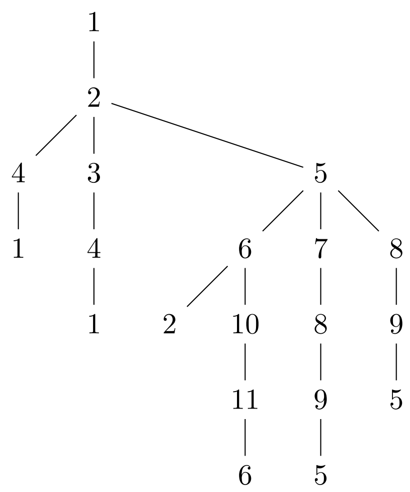


Figure 4.6: The tree generated from the graph of Figure 4.5 by the algorithm described in Section 4.3.1.

Chapter 5

Language Inclusion Problem of Monadic CAs

In this chapter, we introduce our algorithm for solving the language inclusion problem of MCAs, which are defined in Section 4.1. Recall that any CA can be transformed to an equivalent NFA. In particular, the language inclusion problem of MCAs can be transformed in terms of NFAs, so we can apply the solution given in Section 3.3. We present the main idea of our algorithm from which the structure of this chapter follows. The language inclusion problem of MCAs M_1 and M_2 is to decide whether $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$, or, equivalently, $\mathcal{L}(M_1) \cap \mathcal{L}(\overline{M_2}) = \emptyset$ where $\overline{M_2}$ is the complement of M_2. The automaton that recognizes such a language is called a *product automaton* of M_1 and $\overline{M_2}$, written $M_1 \times \overline{M_2}$. Our algorithm is based on building the product automaton and checking whether some final state in the product automaton is reachable and its final condition is satisfiable. We note that our algorithm for solving the inclusion problem of MCAs leads to a different method for solving the inclusion problem of eREs, i.e., whether the language of the first eRE is included in the second eRE. The method is based on transforming the eREs to the MCAs and applying our algorithm for the inclusion problem of MCAs. In Chapter 6, we experimentally evaluate the method based on MCAs and compare them with the method based on NFAs (eREs are transformed into NFAs and applying an algorithm for testing the language inclusion of NFAs).

Before we can construct the product automaton, we need to know how to complement M_2. The complementation of M_2 is based on the determinization of M_2. For that reason, we present in Section 5.1 a determinization algorithm for MCAs [14, Section 4.2]. Checking whether a final state of the product automaton is reachable (and its final condition is satisfiable) is the same as testing whether the language of the product automaton is empty. Similarly, as we accelerate the self-loops in MCAs (see Section 4.1), we need to know how to accelerate the self-loop in determinized MCAs. The accelerations are used to speed up testing whether the language of the product automaton is empty (Section 5.2). In Section 5.3, we give a procedure that builds the product automaton $M_1 \times \overline{M_2}$. Finally, in Section 5.4 we provide a procedure for testing emptiness of the product automaton.

5.1 Determinization of Monadic CAs

A general algorithm for determinization of CAs is developed in [14]. They also provide a more efficient algorithm if the input is an MCA. In this section, we briefly introduce this algorithm for determinization of MCAs (see [14, Section 4.2] for a complete discussion). Since the resulting automata of the algorithm are still somewhat restricted as we show in Section 5.2, we called these automata *determinized monadic* CAs (DMCAs), sometimes they are also called *determinized* MCAs.

Let $M = (Q, C, I, F, \Delta)$ be an MCA. Suppose that q is a counting state. Note that the counter guards on c_q appear only on the transitions leaving q (including the self-loop of q). That is, the value of c_q has no influence on the a-successors of the current configuration of M for any $a \in \Sigma$ if the configuration is not in q. To represent different variants of c_q, we use parameters of the form $c_q[i]$ obtained by indexing c_q by an index i, for $0 \leq i \leq \boldsymbol{max}_q$, while enforcing the invariant $c_q[i] \neq c_q[j]$ whenever $i \neq j$. Recall that the values of c_q are between 0 and $\boldsymbol{max}_q$, thus at most $\boldsymbol{max}_q + 1$ variants of c_q are needed.

Since we need to remember only the variants of c_q if we are in q and no other variants of the different counters than c_q, the states can be represented by the *sphere*:

$$\Psi := \bigvee_{q \in Q'_s} \mathtt{s} = q \vee \bigvee_{q \in Q'_c} \big(\mathtt{s} = q \wedge \bigvee_{0 \leq i \leq \boldsymbol{max}'_q} c_q = c_q[i]\big) \tag{1}$$

for some $Q'_s \subseteq Q_s$, $Q'_c \subseteq Q_c$, and $\boldsymbol{max}'_q \leq \boldsymbol{max}_q$. That is, a sphere Ψ records which states may be reached in the original MCA when Ψ is reached in the determinized MCA and also which variants of the counter c_q may record the value of c_q when q is reached.

From the structure of MCAs it follows that the variants of $c_q[i]$ stay sorted. That is, we have $\alpha(c_q[i]) < \alpha(c_q[j])$ in every configuration α of determinized MCA whenever $i < j$. Since the variants $c_q[i]$ are sorted, it is easy to see that the variant of c_q with the highest index, called the *highest variant*, has the highest value. This, together with the invariant that every c_q is bounded by $\boldsymbol{max}_q$ and mutual distinctness of value of variants of c_q, means that the highest variant is the only one that may satisfy the condition $c_q = \boldsymbol{max}_q$ on the exit transitions or fail the condition $c_q < \boldsymbol{max}_q$ on the self-loop.

Moreover, if the state q is a range counting state, then only the smallest variant of c_q (the one with the smallest index) is important. Intuitively, suppose that we are in the state q with the variants $c_q[i]$ and $c_q[j]$ such that $i < j$. Then every move from q with the variant $c_q[j]$ can be simulated by some move from q with the variant $c_q[i]$, since every exit transition from q has the counter guard equal to $\top$ and the increment self-loop has the counter guard $c_q < \boldsymbol{max}_q$ (note that $c_q[i] < c_q[j]$). Furthermore, if q is a final state, then both variants satisfy the final condition (the final condition is equal to $\top$). Note that the smallest variant of c_q can be always stored in $c_q[0]$.

In the determinization algorithm, we will represent the sphere by a multiset of states. By a slight abuse of notation, we use Ψ for the sphere itself as well as for its multiset representation $\Psi : Q \to \mathbb{N}$. The fact that $\Psi(q) > 0$ means that q is present in the sphere, i.e., $\mathtt{s} = q$ is a predicate in the sphere (1), and for a counting state q, the counters $c_q[0], \ldots, c_q[\Psi(q) - 1]$ are the $\Psi(q)$ variants c_q tracked in the sphere, i.e., $\boldsymbol{max}'_q = \Psi(q)$ in the sphere (1).

Determinization Algorithm of MCAs

Let $M = (Q, C, I, F, \Delta)$ be an MCA. The algorithm, which is introduced in [14, Section 4.2], produces a language equivalent DMCA $D = (Q^D, C^D, I^D, F^D, \Delta^D)$ in the following way. The high-level description of the algorithm is written in Algorithm 8.

The initial sphere Ψ_I assigns 1 to all initial states in M (and 0 to all non-initial states). The initial condition I^D ensures that we start from the initial sphere Ψ_I with initialized

Algorithm 8: MCA determinization algorithm

Input : An MCA $M = (Q, C, I, F, \Delta)$
Output : A DMCA $D = (Q^D, C^D, I^D, F^D, \Delta^D)$ such that $\mathcal{L}(M) = \mathcal{L}(D)$

1 $Q^D \leftarrow \emptyset$; $\Delta^D \leftarrow \bot$;
2 $\Psi_I \leftarrow \{q \mapsto 1 \mid q$ is an initial state in $M\}$;
3 $I^D \leftarrow \mathtt{s} = \Psi_I \wedge \bigwedge_{q \mapsto 1 \in \Psi_I} c_q[0] = 0$;
4 $Worklist \leftarrow \{\Psi_I\}$;
5 **while** $Worklist \neq \emptyset$ **do**
6   pick and remove Ψ from $Worklist$;
7     $Q^D \leftarrow Q^D \cup \{\Psi\}$;
8   **foreach** $\mu \in Minterms(\Delta_\Psi)$ **do**
9   compute the exit transition $\Psi \xrightarrow{\{\sigma,g,f\}} \Psi'$;
10         $\Delta^D \leftarrow \Delta^D \vee \Psi \xrightarrow{\{\sigma,g,f\}} \Psi'$;
11   **if** $\Psi' \notin Q^D$ **then**
12             $Worklist \leftarrow Worklist \cup \{\Psi'\}$;
13 $C^D \leftarrow$ all variants of counters found in Q^D;
14 $F^D \leftarrow \bigvee_{\Psi \in Q^D} \mathtt{s} = \Psi \wedge \exists\, C, \mathtt{s} : (\Psi \wedge F)$;
15 $I^D \leftarrow ground(I^D)$; $\Delta^D \leftarrow ground(\Delta^D)$;
16 **return** $(Q^D, C^D, I^D, F^D, \Delta^D)$;

values of counters— I^D assigns 0 to $c_q[0]$ for each initial counting state q in M (Lines 2, 3). The set $Worklist$ stores all spheres which are not processed yet (we do not compute the exit transitions from these spheres). Thus, initially, only the sphere Ψ_I is in $Worklist$ (Line 4). Until $Worklist$ is empty, we pick and remove the sphere Ψ from $Worklist$. Moreover, this sphere is added to Q^D (Lines 5–7).

Let Δ_Ψ denote the set of transitions of M originating from the states q with $\Psi(q) > 0$. We remove the counter guard $c_q < \boldsymbol{max}_q$ from every self-loop of an exact counting state q in Δ_Ψ (since this counter guard has no semantic effect, i.e., the language of M remains the same).

Subsequently, we compute the set of minterms of the set of symbol and counter guard formulae of the transitions in Δ_Ψ, we denote this set by $Minterms(\Delta_\Psi)$. Each minterm $\mu \in Minterms(\Delta_\Psi)$ then corresponds to a transition $\Psi \xrightarrow{\{\sigma,g,f\}} \Psi'$ of D (Line 8). The symbol and counter guard formulae σ and g, assignments formula f, and the target sphere Ψ' are constructed from μ as follows (Line 9).

First, the symbol and counter guards σ and g are obtained from the minterm μ by replacing every occurrence of c_q by $c_q[\Psi(q)]$ for each $q \in Q_c$. In other words, we replace every occurrence of c_q by the highest variant of c_q (recall that only the highest variant of c_q may satisfy the condition on the exit transition or fail the condition on the increment self-loop). Second, we initialize the target sphere Ψ' as the empty multiset $\{q \mapsto 0 \mid q \in Q\}$. The set Δ_μ consists of all transitions from Δ_Ψ that are compatible with the minterm μ. Third, the assignment formula f is obtained and the target sphere Ψ' is modified by processing the transitions of Δ_μ in the following three steps.

Step 1 (simple states). For every simple states q with an entry transition in Δ_μ, we define $\Psi'(q) = 1$.

Step 2 (increment self-loops). For every exact counting state q with the increment transition in Δ_μ, we set $\Psi'(q)$ to $\Psi(q) - 1$ if an exit transition of q is in Δ_μ, and to $\Psi(q)$

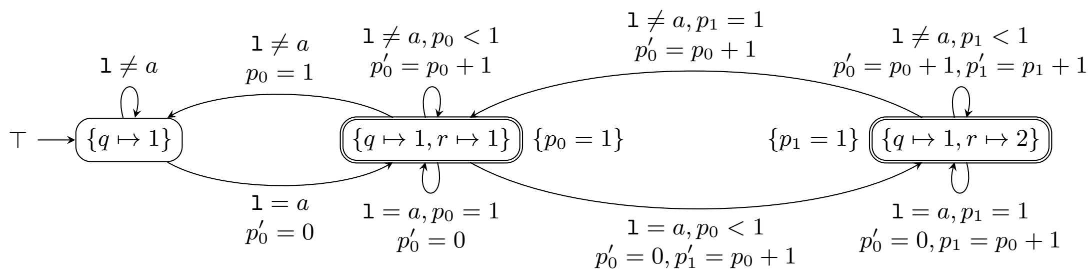


Figure 5.1: The DMCA generated from the MCA in Figure 4.1 for $k = 1$ by the determinization algorithm of MCA (Section 5.1) [14].

otherwise. For every range counting state q with the increment transition in Δ_μ, we set $\Psi'(q)$ to 1. Then the assignment formula f is the conjunction of $c_q[i]' = c_q[i] + 1$ for each $0 \leq i < \Psi'(q)$, since the variants that take the self-loop are incremented.

Step 3 (entry transitions). For each counting state q with an entry transition in Δ_μ, $\Psi'(q)$ is incremented by 1 and the assignment $c_q[0]' = 0$ of the fresh variant of c_q is added to f. If the increment causes that the value of $\Psi'(q)$ exceeds $\boldsymbol{max}_q + 1$, then the whole transition is discarded, since c_q cannot have more that $\boldsymbol{max}_q + 1$ variants of c_q. If q is an exact counting state, then f must be updated to preserve the invariant of sorted and unique values of c_q: all increments of variant c_q (except the one added in this step) are right-shifted to make space for the fresh variant—each conjunct $c_q[i]' = c_q[i] + 1$ in f is replaced by $c_q[i+1]' = c_q[i] + 1$. If q is a range counting state and the assignment $c_q[0]' = c_q[0] + 1$ is present in f, then we remove this assignment from f, since only the smallest variant of q is important and 0 is the smallest possible variant.

After the symbol, counter, assignment formulae, and the target sphere are constructed, we added the transition $\Psi \text{-}\{\sigma,g,f\}\!\!\rightarrow \Psi'$ to Δ^D (Lines 10). If the target sphere Ψ' is new (i.e., $\Psi' \notin Q^D$), then we need to process the exit transitions from Ψ'. Thus we add Ψ' to $Worklist$ (Lines 11, 12).

Finally, we collect the set C^D of all variants of counters of D used in the spheres of Q^D (Line 13). The final condition F^D of D considers all spheres in Q^D by restricting them to valuation where the original final formula F is satisfied. Moreover, we quantify out the original counters (this is the meaning of the symbol $\exists$). In this way, the final constraints in the final condition get translated to constraints over counters in C^D (Line 14). The constructed automaton D can be nondeterministic due to unused and unconstrained counters. This nondeterminism is resolved by the function *ground*, which we apply on I^D and Δ^D (Line 15). The application of *ground* on I^D adds conjuncts of the from $c = 0$ for every $c \in C^D$ that is so far unconstrained in I^D. And the application of *ground* on Δ^D modifies every transition φ as follows. The function *ground* adds conjuncts of the from $c = 0$ for every $c \in C^D$ that is so far unconstrained in φ. Moreover, *ground* introduces a reset $c' = 0$ for every counter c that is so far not assigned in φ. If we apply Algoritm 8 on the MCA in Figure 4.1 for $k = 1$, then we obtain the DMCA in Figure 5.1.

5.2 Structure of Determinized Monadic CAs

In the last section, we introduced the determinized MCAs (DMCAs). The goal of this section is to investigate their structure. If the structure of DMCAs is known, then we can use this knowledge to accelerate the self-loops of states in the DMCAs, i.e., how the values of the variants of counters are changed if the self-loop is executed several times. The structure of DMCAs is not given by the definition like the structure of MCAs but it follows from the determinization algorithm, which was introduced in the last section (see Algorithm 8).

Determinized MCAs may not be again MCAs as witnessed by the DMCA in Figure 5.1. Moreover, a state of DMCAs may have several self-loops (see Figure 5.1) in the contrast to MCAs, where each state has at most one self-loop. Nevertheless, the fact that a DMCA may not be an MCA, the structure of DMCAs is still somewhat restricted as we show in this section. We are mostly interested in the forms of the self-loops in DMCAs.

Let $M = (Q, C, I, F, \Delta)$ be an MCA and $D = (Q^D, C^D, I^D, F^D, \Delta^D)$ be a normalized DMCA that is obtained from M by Algorithm 8. Let $R = \{q_1 \mapsto 1, q_2 \mapsto 1, \cdots, q_n \mapsto k\} \in Q^D$ be a sphere where only q_n is a counting state (q_i is a simple state for each $1 \leq i \leq n-1$) and $k > 0$. We assume that all increment and exit transitions φ_i from q_i have $sym(\varphi_i) = \sigma$. Later we show that this assumption is not necessary. Let $\varphi = R \text{-}\{\alpha\}\text{→} P \in [\![\Delta^D]\!]$ be a transition. The structure of α depends on whether q_n is a range counting state or q_n is an exact counting state. Moreover, we define the *label* of a transition $R \text{-}\{\alpha\}\text{→} P$ to be α.

Range Counting State

Suppose that q_n is a range counting state. By the determinization algorithm we must have $k = 1$. Thus R is equal to $\{q_1 \mapsto 1, q_2 \mapsto 1, \cdots, q_n \mapsto 1\}$. In Figure 5.2 (a), we see a part of the original automaton M (it follows from the structure of the sphere R). The determinization algorithm computes the following set of minterms (here we use the assumption that all exit transitions from q_i in R have the same symbol guards)

$$\Phi = \{\sigma \wedge c_q < \boldsymbol{max}_{q_n}, \neg\sigma \wedge c_q < \boldsymbol{max}_{q_n}, \sigma \wedge c_q \geq \boldsymbol{max}_{q_n}, \neg\sigma \wedge c_q \geq \boldsymbol{max}_{q_n}\}.$$

The possible outgoing transitions $R \text{-}\{\alpha\}\text{→} P$ have labels of the following forms, where δ is either σ, $\neg\sigma$, or $\top$:

(Ia) $\alpha = (\delta, c_q[0] < \boldsymbol{max}_{q_n}, c_q[0]' = c_q[0] + 1)$,

(IIa) $\alpha = (\delta, c_q[0] < \boldsymbol{max}_{q_n}, c_q[0]' = 0)$,

(IIIa) $\alpha = (\delta, c_q[0] \geq \boldsymbol{max}_{q_n}, c_q[0]' = c_q[0] + 1)$,

(IVa) $\alpha = (\delta, c_q[0] \geq \boldsymbol{max}_{q_n}, c_q[0]' = 0)$.

Note that (IIIa) cannot be a label of a self-loop on R (we can be sure that $R \neq P$). The reason is the following: if $c_q[0] \geq \boldsymbol{max}_{q_n}$, then the counter guard of the self-loop of q_n in M is unsatisfiable. Moreover, there is no entry transition coming to q_n from q_i, for $1 \leq i \leq n-1$, otherwise we would have $c_q[0]' = 0$ by the algorithm. Hence the target sphere contains $q_n \mapsto 0$. Therefore the source and target sphere of the transitions must be different (the source sphere has $q_n \mapsto 1$), i.e., this transition cannot be a self-loop on R.

In general, all increment and exit transitions φ_i from q_i have $sym(\varphi_i) = \sigma_i$. Then the set of minterms Φ is computed from the set $\{\sigma_1, \ldots, \sigma_n, c_q < \boldsymbol{max}_{q_n}\}$. Again, all self-loops of R are of the form (Ia), (IIa), or (IVa) where $\delta = \bigwedge_{i=1}^{n} \delta_i$ such that δ_i is either σ_i or $\neg\sigma_i$. Since D is normalized, R has at most three self-loops, each of which is of the form (Ia), (IIa), or (IVa). It should be mentioned that not all combinations of the forms are allowed, because of the determinism of D. For example, if R contains two self-loops of the form (Ia) and (IIa) where δ is the same in both, then it contradicts that D is deterministic.

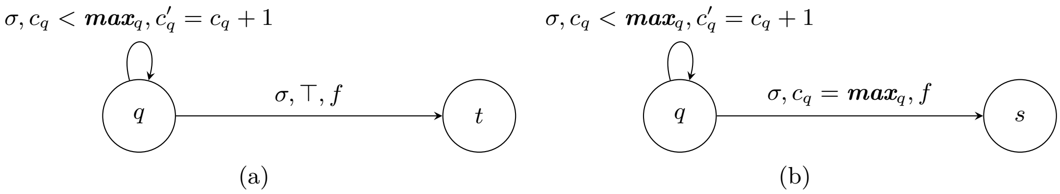


Figure 5.2: On the left is a part of some MCA with a range counting state q. We see the structure of the self-loop of q and the exit transition from q. In general, there is more than one exit transition from q, but always there is only one self-loop of q. On the right is depicted the same situation for an exact counting state q.

Exact Counting State

We repeat the process from the preceding subsection for the case if q_n is an exact counting state. Now, the value of k in R is an arbitrary positive integer. In Figure 5.2, we see a part of the original automaton M. Using the determinization algorithm, we compute the following set of minterms

$$\Phi = \{\sigma \wedge c_q = \boldsymbol{max}_{q_n}, \neg\sigma \wedge c_q = \boldsymbol{max}_{q_n}, \sigma \wedge c_q \neq \boldsymbol{max}_{q_n}, \neg\sigma \wedge c_q \neq \boldsymbol{max}_{q_n}\}.$$

The possible outgoing transitions $R\text{-}\{\alpha\}\!\!\rightarrow\! P$ have labels of the following forms, where δ is either σ, $\neg\sigma$, or $\top$:

(Ib) $\alpha = (\delta, c_q[\Psi(q_n) - 1] = \boldsymbol{max}_{q_n}, \bigwedge_{i=0}^{\Psi(q_n)-2} c_q[i]' = c_q[i] + 1)$,

(IIb) $\alpha = (\delta, c_q[\Psi(q_n) - 1] \neq \boldsymbol{max}_{q_n}, \bigwedge_{i=0}^{\Psi(q_n)-1} c_q[i]' = c_q[i] + 1)$,

(IIIb) $\alpha = (\delta, c_q[\Psi(q_n) - 1] = \boldsymbol{max}_{q_n}, c_q[0]' = 0 \wedge \bigwedge_{i=0}^{\Psi(q_n)-2} c_q[i+1]' = c_q[i] + 1)$,

(IVb) $\alpha = (\delta, c_q[\Psi(q_n) - 1] \neq \boldsymbol{max}_{q_n}, c_q[0]' = 0 \wedge \bigwedge_{i=0}^{\Psi(q_n)-1} c_q[i+1]' = c_q[i] + 1)$.

Note that (Ib) cannot be a label of a self-loop of R because from the structure of the counter update we know that there is no incoming transition to q_n from q_i, for $1 \leq i \leq n-1$. Moreover, the counter guard of the self-loop of q_n is unsatisfiable. It follows from the determinization algorithm that the target sphere contains $q_n \mapsto 0$ (the source sphere has $q_n \mapsto k$ where $k > 0$). Therefore the source and the target spheres are different.

It is not hard to see that (IVb) cannot be a label of the self-loop of R. The justification is the following: the label α creates a new variant of the counter c_q and the highest variant of c_q satisfies the counter guard of the self-loop of q. So the target sphere works with one more variant of c_q than the source sphere, i.e., the target sphere contains $q_n \mapsto k+1$. Therefore the source and the target spheres are different (the source sphere has $q_n \mapsto k$).

Analogously as for the range counting states, the assumption that all outgoing transitions from q_i have $sym(q_i) = \sigma$ is not necessary. Thus R has at most two self-loops, each of which is of the form (IIb) or (IIIb). Again, not all combinations of self-loops on R are possible because D is deterministic.

Acceleration of Self-loops in a DMCA

In the preceding two subsections, we investigated the forms of self-loops on a sphere $R = \{q_1 \mapsto 1, q_2 \mapsto 1, \cdots, q_n \mapsto k\}$ containing only single counting state. We found out that there are five possible forms of self-loops on R depending on whether q_n is a range or an exact counting state. Because some forms are special cases of other cases, it is reasonable to reduce the number of such forms to the minimum.

First, note that $c_q[i] < \boldsymbol{max}_q$ and $c_q[i] \neq \boldsymbol{max}_q$ are equivalent, for any i, because there is not a possibility to have $c_q[i] > \boldsymbol{max}_q$ by the definition of CAs. Second, $c_q[i] \geq \boldsymbol{max}_q$ and $c_q[i] = \boldsymbol{max}_q$ are equivalent by the same reason. It follows that (Ia) is a special case of (IIb), and (IVa) is a special case of (IIIb). Finally, we write the possible forms (without duplicates) of self-loops on R:

(I) $\alpha = (\delta, c_q[0] < \boldsymbol{max}_{q_n}, c_q[0]' = 0)$,

(II) $\alpha = (\delta, c_q[\Psi(q_n) - 1] < \boldsymbol{max}_{q_n}, \bigwedge_{i=0}^{\Psi(q_n)-1} c_q[i]' = c_q[i] + 1)$,

(III) $\alpha = (\delta, c_q[\Psi(q_n) - 1] = \boldsymbol{max}_{q_n}, c_q[0]' = 0 \wedge \bigwedge_{i=0}^{\Psi(q_n)-2} c_q[i+1]' = c_q[i] + 1)$.

Since we already know the structure of the self-loops of the states in DMCAs, we are ready to give the acceleration formulae of such self-loops. Note that there is no need to accelerate the self-loops of the form (I), since every execution of the self-loop set $c_q[0]$ to 0. In other words, the acceleration formula is the same as the counter guard and assignment formula of α. The self-loops of the form (II) have a similar structure as the increment self-loops in MCAs (see Section 4.1). The only difference is that the self-loops of the form (II) increment more variants of the counter. Thus the acceleration formula of self-loops of the form (II) is a simple extension of the acceleration formula of the increment self-loops in MCAs. The acceleration formula can look like $\exists k : \big(0 \leq k \leq \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{\Psi(q)-1} (c_q[i]' = c_q[i] + k \wedge c_q[i]' \leq \boldsymbol{max}_q)\big)$. Using the fact that the highest variant of c_q has the highest value, we obtain the final version of the acceleration formula of self-loops of the form (II):

$$\exists k : \Big(0 \leq k \leq \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{\Psi(q)-1} \big(c_q[i]' = c_q[i] + k\big) \wedge c_q[\Psi(q) - 1]' \leq \boldsymbol{max}_q\Big). \tag{5.1}$$

Note that if we have $\Psi(q) = 1$ in the formula (5.1), then we obtain the acceleration formula of the increment self-loop in MCAs.

The acceleration formula of the self-loops of the form (III) is a little bit more complicated. First, note that if $\Psi(q) - 1 = \boldsymbol{max}_q$, then the self-loop do not have to be accelerated. The reason is the following: we know that $c_q[\Psi(q) - 1] = \boldsymbol{max}_q$. Since every variant of c_q has distinct (nonnegative) value, we know that $c_q[i+1] = c_q[i]+1$ for every $1 \leq i \leq \Psi(q)-2$ and $c_q[0] = 0$. If the previous is not true, then there are two distinct variants having the same values, which contradict the invariant of unique value of variants. Thus after executing the self-loop of the form (III) we obtain the same value as before the execution. Thus suppose that $\Psi(q) < \boldsymbol{max}_q$. It is not hard to see that after $k > 0$ iterations of the self-loop of the form (III) the k-th lowest variant of c_q has the value $k - 1$ (note that the k-th lowest variant is the one with the index $k - 1$), the $(k - 1)$-th lowest variant of c_q has the value $k - 2$, and so on. The new value of the variant with the index $i > k$ is obtained from the value of the variant $c_q[i-k]+k$, since the value of $c_q[i-k]$ is k times shifted to the right and in each shift the variant is increased by one. It remains to determine how many times such a self-loop can be executed. Intuitively, if the self-loop is executed once, then $c_q[\Psi(q) - 1] = \boldsymbol{max}_q$. If twice, then we must have $c_q[\Psi(q) - 2] + 1 = \boldsymbol{max}_q$, since after one execution of the self-loop the highest variant has the value $c_q[\Psi(q) - 2] + 1$. In general, if the self-loop is executed k times, then $c_q[\Psi(q) - k] + k - 1 = \boldsymbol{max}_q$. Moreover, there is always an option that the self-loop does not have to be executed. In this case, the new values of variants are the same as the old ones. This all can be write into a single formula:

$$\exists k : \Big(0 < k \leq \Psi(q) \wedge \big(c_q[\Psi(q) - k] + k - 1\big) = \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{k-1} c_q[i]' = i \\ \wedge - \bigwedge_{i=k}^{\Psi(q)-1} c_q[i]' = c_q[i-k] + k\Big). \tag{5.2}$$

General Form of Sphere in a DMCA

In the previous sections, we discussed the forms of self-loops of a sphere where only one state in the sphere was counting. In general, the sphere may consist of several counting states. Let $R = \{q_1 \mapsto k_1, \dots, q_j \mapsto k_j, q_{j+1} \mapsto 1, \dots, q_n \mapsto 1\} \in Q^D$ be a sphere where q_i are counting states (either range or exact) with $k_i > 0$ for $1 \leq i \leq j$ and the rest are simple states. Suppose that $R\text{-}\{\alpha\}\!\mapsto P \in [\![\Delta^D]\!]$, we again investigate the forms of the label α and identify which of them can be a self-loop on R.

The symbol and counter guards of α originate from the set of minterms Φ, which are computed from the set $S = \{c_1 \otimes \boldsymbol{max}_{q_1}, \dots, c_j \otimes \boldsymbol{max}_{q_j}, \sigma\}$, where $\otimes$ is either $<$ or $=$ depending on whether q_i is a range or an exact counting state, respectively. Thus a minterm $\mu \in \Phi$ is of the form $\delta \wedge \bigwedge_{i=1}^{j} \varphi_{q_i}$ where δ is either σ or $\neg\sigma$ and φ_{q_i} is $c_i \otimes \boldsymbol{max}_{q_i}$ or its negation. The counter assignment formula of α with the guard μ is computed by precessing the transitions from Δ_R that are compatible with μ (see Section 5.1). Using the determinization algorithm, the counter assignment formula can be written in the form $\psi_{q_1} \wedge \dots \wedge \psi_{q_j}$, where ψ_{q_i} updates only the variants of counter c_{q_i}. Thus the label α of R can be divided in the conjuncts as $\delta \wedge \varphi_{q_1} \wedge \dots \wedge \varphi_{q_j} \wedge \psi_{q_1} \cdots \wedge \psi_{q_j}$ where $\delta \wedge \varphi_{q_i} \wedge \psi_{q_i}$ is of one the forms (Ia)–(IVa), (Ib)–(IVb). Let Ω_i denote the formula $\delta \wedge \varphi_{q_i} \wedge \psi_{q_i}$.

If any Ω_i is not of the form (I)–(III), then the transition $R\text{-}\{\alpha\}\!\mapsto P$ cannot be a self-loop of R (for one of the reasons described in the above sections). Otherwise, each Ω_i is of the form (I)–(III) and $R\text{-}\{\alpha\}\!\mapsto P$ may be a self-loop. Suppose that $R\text{-}\{\alpha\}\!\mapsto P$ is a self-loop, i.e., $P = R$. From the preceding paragraph, we know that α can be divided in $\delta \wedge \varphi_{q_1} \wedge \dots \wedge \varphi_{q_j} \wedge \psi_{q_1} \cdots \wedge \psi_{q_j}$ where Ω_i is of one the forms (I)–(III). If there is a self-loop with the label Ω_i, then from the previous sections the acceleration formula of a such self-loop is known. Then the acceleration formula of $R\text{-}\{\alpha\}\!\mapsto R$ is the conjunction of the partial acceleration formula of Ω_i while ensuring that the bounded variables by the existential quantifiers are the same. Formally, let χ_i be the acceleration formula of Ω_i with the bounded variable k_i. The acceleration formula of $R\text{-}\{\alpha\}\!\mapsto P$ is $\bigwedge_{i=1}^{n} \chi_i \wedge k_1 = k_2 = \dots = k_n$ (we assume that each variable k_i is bounded by the existential quantifier also in the second conjunct).

Lastly, we note that the assumption that all transitions φ_i have the same symbol guard is not necessary. The only difference is that the set S contains more σ_i for each $sym(\varphi_i) = \sigma_i$. Thus we obtain more minterms, but it has no impact on the counter guards and updates (the detailed consequences are described in the section *Range counting state* above).

5.3 Product Construction of MCA and DMCA

Let M_1, M_2 be MCAs. In this section, we give an algorithm for building the *product automaton* of M_1 and the complement of M_2, denoted by $M_1 \times \overline{M_2}$. The language of such an automaton is $\mathcal{L}(M_1) \cap \mathcal{L}(\overline{M_2})$. The states of the product automaton are sometimes called *product-states.*

Before we start building the product automaton, we need to compute the complement of M_2, written $\overline{M_2}$, which accepts a string $w \in \Sigma^*$ iff M_2 does not accept w, i.e., $\mathcal{L}(\overline{M_2}) = \overline{\mathcal{L}(M_2)}$. The main idea is to make the states in $\overline{M_2}$ final only if the states are not final in M_2 and vice versa. This approach works only if M_2 is deterministic and complete, since we need to have exactly one w-successor of α for all configurations α and strings $w \in \Sigma^*$. How to make M_2 deterministic and complete is described in Sections 5.1 and 2.2.3, respectively. Using this methods, we obtain a complete determinized MCA $M_2' = (Q^D, C^D, I^D, F^D, \Delta^D)$ with the same language as M_2. Now, to complement M_2' we just complement its final condition, i.e., $\overline{M_2} = (Q^D, C^D, I^D, \neg F^D, \Delta^D)$. We define the function *complement* that takes an MCA and produces a DMCA as described above.

Example 5.1. Let $D = (Q, C, I, F, \Delta)$ be the same DMCA as in Figure 5.1. We compute the complement of D by the method described above. Since D is already deterministic, we need to only make D complete. Using the method given in Section 2.2.3 we obtain $D' = (Q \cup \{q_{sink}\}, C, I, F, \Delta')$ where $\Delta' : \Delta \vee q_{sink} \xrightarrow{\{\top,\top,\top\}} q_{sink} \vee \{q \mapsto 1, r \mapsto 2\} \xrightarrow{\{\mathtt{1} \neq a, p_1 < 1, \top\}} q_{sink}$. Then we complement the final formula of D', which can be equivalently written as

$$\neg F : \mathtt{s} = \{q \mapsto 1\} \vee \big(\mathtt{s} = \{q \mapsto 1, r \mapsto 1\} \wedge p_0 \neq 1\big) \vee \mathtt{s} = \{q \mapsto 1, r \mapsto 2\} \wedge \big(p_1 \neq 1 \vee \mathtt{s} = q_{sink}\big).$$

Algorithm 9 builds the product automaton $M_1 \times \overline{M_2}$ as follows. We are given MCAs M_1 and M_2 with distinct sets of states and counters. First, we complement M_2 using the function *complement* (Line 1). The product-states of the output automaton $N = (Q, C, I, F, \Delta)$ are pairs (q, R), where $q \in Q_1$ and $R \in Q^D$, i.e., $Q \subseteq Q_1 \times Q^D$. The set of counters of N is $C \subseteq C_1 \cup C^D$ (some counter might not be needed if its corresponding state is not reachable, see below the function *ground*). The initial formula I of N labels pairs of states as initial if both states are also initial in M_1 and $\overline{M_2}$, respectively (Line 2). Formally, we transform $I = I_1 \wedge I^D$ into disjunctive normal formal such that each part of disjuncts of the form $\mathtt{s} = q \wedge \mathtt{s} = R$ is replaced by $\mathtt{s} = (q, R)$.[1] The initial values of counters are then the combinations of initial values of M_1 and $\overline{M_2}$. This transformation is denoted by dnf, so $I = dnf(I_1 \wedge I^D)$. On Line 3, we initialize the set Q and $Worklist$ by the product-states that appear in I. The rest of the product automaton is built by processing the states (q, R) from $Worklist$ and creating new states (q', R') that originate by combining the target states of outgoing transitions from q and R. In detail, until $Worklist$ is empty, we pick and remove the product-state (q, R) from $Worklist$ (Lines 4, 5). The outgoing transitions from (q, R) are created by combing the label α_1 and α_2 of outgoing transitions from q and R, respectively (Line 6). The outgoing transitions are combined only if the conjunction of $sym(\varphi_1)$ and $sym(\varphi_2)$ is satisfiable (Line 9). If the state (q', R') generated from (q, R) by the transition with the label $\alpha_1 \wedge \alpha_2$ is new, then we add (q', R') to both $Worklist$ and Q (Lines 11, 12). Moreover, we add $(q, R) \xrightarrow{\{\alpha_1 \wedge \alpha_2\}} (q', R')$ to Δ (Line 13). The set of counters C in N consists of used counters from $C_1 \cup C^D$. That is, we take $C_1 \cup C^D$ and remove all counters that do not appear in any guards of I and Δ (Line 14), which is performed by the function *ground*. The final formula F of N is computed analogously as I (Line 15), with the difference that we need to remove from F product-states that are not in Q, which is the purpose of *ground* in this case.

[1] If we have $\varphi := (\mathtt{s} = q) \wedge \psi \wedge (\mathtt{s} = R)$, then we transform φ to $(\mathtt{s} = q) \wedge (\mathtt{s} = R) \wedge \psi$ by using the commutative law. Similarly, if $\varphi := (\mathtt{s} = R) \wedge \psi \wedge (\mathtt{s} = q)$ we can use the commutative law to get the correct order of $\mathtt{s} = q$ and $\mathtt{s} = R$.

Algorithm 9: Product automaton of an MCA and a DMCA

Input : MCAs $M_1 = (Q_1, C_1, I_1, F_1, \Delta_1)$, $M_2 = (Q_2, C_2, I_2, F_2, \Delta_2)$ with $Q_1 \cap Q_2 = C_1 \cap C_2 = \emptyset$
Output : A CA $N = M_1 \times \overline{M_2}$ such that $\mathcal{L}(N) = \mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)}$

$(Q^D, C^D, I^D, \neg F^D, \Delta^D) \leftarrow complement(M_2)$;
$I \leftarrow dnf(I_1 \wedge I^D)$; $\Delta \leftarrow \emptyset$;
$Q \leftarrow Worklist \leftarrow \{(q, R) \mid \mathtt{s} = (q, R) \text{ appears in } I\}$;
while $Worklist \neq \emptyset$ **do**
  pick and remove (q, R) from $Worklist$;
  foreach $\varphi_1 = q \xrightarrow{\alpha_1} q' \in \Delta_1$ and $\varphi_2 = R \xrightarrow{\alpha_2} R' \in \Delta^D$ **do**
    Let $\sigma_1 = sym(\varphi_1)$;
    Let $\sigma_2 = sym(\varphi_2)$;
    if $IsSat(\sigma_1 \wedge \sigma_2)$ **then**
      if $(q', R') \notin Q$ **then**
        $Worklist \leftarrow Worklist \cup \{(q', R')\}$;
        $Q \leftarrow Q \cup \{(q', R')\}$;
      $\Delta \leftarrow \Delta \cup \{(q, R) \xrightarrow{\alpha_1 \wedge \alpha_2} (q', R')\}$;
$C \leftarrow ground(C_1 \cup C^D)$;
$F \leftarrow ground(dnf(F_1 \wedge \neg F^D))$;
return (Q, C, I, F, Δ);

We note that some pairs of states in the product automaton might not be reachable, because we only combine transitions $\varphi_1 \in \Delta_1$ and $\varphi_2 \in \Delta^D$ for which $sym(\varphi_1) \wedge sym(\varphi_2)$ is satisfiable and completely ignore the counter guards, which can cause that the transition is unreachable—this is purpose of the next section.

Self-loops in the Product of an MCA and a DMCA

We already know how to accelerate the self-loops on the general states in MCAs and DMCAs. Finally, we will take a look on how the self-loops in the product automaton of an MCA and a DMCA can be accelerated. Suppose that M_1, M_2 are normalized MCAs. Let (q, R) be a product-state in $M_1 \times \overline{M_2}$. The state (q, R) has at least one self-loop iff q has one self-loop in M_1 and R has at least one self-loop in $\overline{M_2}$. Thus the number of self-loops in $M_1 \times \overline{M_2}$ is equal to the number of self-loop in $\overline{M_2}$.

Let $(q, R) \xrightarrow{\alpha_1 \wedge \alpha_2} (q, R)$ be a self-loop in $M_1 \times \overline{M_2}$ such that $q \xrightarrow{\alpha_1} q$ and $R \xrightarrow{\alpha_2} R$ are self-loops in M_1 and $\overline{M_2}$, respectively. From the last section we know the acceleration formulae φ_1 and φ_2 of the self-loops $q \xrightarrow{\alpha_1} q$ and $R \xrightarrow{\alpha_2} R$, respectively. Since the formula α_1 updates different counters than α_2, we can use the same approach as in the acceleration of the self-loop on the general sphere in DMCAs. That is, the acceleration formula of $(q, R) \xrightarrow{\alpha_1 \wedge \alpha_2} (q, R)$ is conjunction of the partial acceleration formulae φ_1 and φ_2 while enforcing that the bounded variables in φ_1 and φ_2 have the same value. Formally, let k_1 be a bounded variable in φ_1 and let k_2 be any bounded variable in φ_2 (in general, φ_2 has several bounded variable). The acceleration formula of $(q, R) \xrightarrow{\alpha_1 \wedge \alpha_2} (q, R)$ is $\varphi_1 \wedge \varphi_2 \wedge k_1 = k_2$ (we assume that the variables k_1 and k_2 are bounded by the existential quantifier also in the last conjunct).

5.4 Language Inclusion Algorithm for MCAs

The main idea of the algorithm for testing language inclusion of MCAs has been already developed. That is, we are given two MCAs M_1, M_2 and the question whether $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$. To answer this question we build the product automaton $M_1 \times \overline{M_2}$ and search for a reachable final state with a satisfiable final condition. If a final reachable state is found (with a satisfiable final condition), then it means that there is a string such that $w \in \mathcal{L}(M_1)$ and $w \notin \mathcal{L}(M_2)$, i.e., $\mathcal{L}(M_1) \nsubseteq \mathcal{L}(M_2)$. To compute the product automaton, we use Algorithm 9. It remains to provide an algorithm for testing reachability of states—this is the purpose of this section.

For a state q, the formula β_q denotes the possible known values of counters if q is reached. For example, if $\beta_q = \exists k : (0 \leq k \leq 5 \wedge c_q = k)$, then the possible values of c_q if q is reached are represented by the set $[\![\beta_q]\!] = \{0, 1, 2, 3, 4, 5\}$. Let φ be a formula, and let $C = \{c_1, \ldots, c_n\}$ be the free variables in φ. The *projection* of φ on C is the formula $\exists c_1, \ldots, \exists c_n : \varphi$.

In Section 5.3, we describe how the self-loops on the states in the product automaton can be accelerated. In fact, we give the acceleration formula that describes how the counters are changed if the self-loop is executed any number of times. But how to obtain the new values of counters? We present a more general technique. Let $\varphi = (q, R) \text{-}\{\alpha\}\text{→} (q', R')$ be a transition. We describe how $\beta_{(q',R')}$ is updated if the transition φ is used. The formula $\beta_{(q,R)}$ denotes the possible values of counters if (q, R) is reached. The formula α describes for which values of counters the transition is satisfiable, and if so then how the values are changed. Thus the formula $\beta_{(q,R)} \wedge \alpha$ restricts the values of $\beta_{(q,R)}$ for which the transition with the label α is satisfiable and only these values the formula updates. In such a formula, there are two types of counters: *unprimed* and *primed* (e.g., c is unprimed and c' is primed). The unprimed counters denote old (or current) values, and the primed counters denote the new (or future) values. To obtain the new values in terms of unprimed counters we proceed as follows. First, we make the projection of $\beta_{(q,R)} \wedge \alpha$ on all unprimed counters used in the formula and then we eliminate all existential quantifiers. Now in the unprimed counters are new values. Second, we replace each primed counter by a corresponding unprimed counter. Suppose that the resulting formula is ψ. Then the formula $\beta_{(q',R')}$ is updated by setting $\beta_{(q',R')} := \beta_{(q',R')} \vee \psi$, since we want to retain the previous known values of counters if (q', R') is reached. This process is demonstrated in Example 5.2. Moreover, we define the function *unprime* that takes a formula φ and replaces every primed counter in φ by its corresponding unprimed counter. Thus we can write $\mathit{unprime}(\mathit{projection}(\beta_{(q,R)} \wedge \alpha))$ to denote the new values of $\beta_{(q',R')}$ if the outgoing transition (possibly self-loop) with the label α from (q, R) to (q', R') is used (the projection is implicitly on all unprimed counters in α).

Since we do not exclude the possibility of $(q, R) = (q', R')$, this process is also applicable for updating the values of $\beta_{(q,R)}$ by using the self-loops on (q, R), but it is strongly inefficient (see Example 5.2). We are interested in how the values of counters are changed after any possible number of executions and not only by a single execution. Of course we can apply this technique several times (we refer to this approach as *trivial acceleration*), but it is sufficient to replace α by its acceleration formula (the acceleration exactly describes what happens if the self-loops are executed several times) and use the approach in the last paragraph.

Example 5.2. Let M_1 be the the same MCA as in Figure 5.2 (b) with the initial state q and the initial condition $I : \mathtt{s} = q \wedge c_q = 0$. For simplicity, suppose that M_2 is a one-state

MCA such that $\mathcal{L}(M_2) = \Sigma^*$. Then the product automaton $M_1 \times \overline{M_2}$ has the same structure as M_1. Let $n = \boldsymbol{max}_q$, we demonstrate the process of updating values of β_q.

The initial formula gives $\beta_q = (c_q = 0)$. First, we use the trivial acceleration. We take $\beta_q \wedge (c_q < n \wedge c'_q = c_q + 1)$ and apply projection on c_q followed by the *unprime* function. The projection on c_q results in the formula $c'_q = 1$ and the application of *unprime* gives $c_q = 1$. The new value of β_q is then $c = 0 \vee c = 1$. In the next step, we proceed exactly as in the previous step. We take $\beta_q \wedge (c_q < n \wedge c'_q = c_q + 1)$ and apply projection on c_q followed by the *unprime* function. The resulting formula is $c_q = 1 \vee c_q = 2$. The new value of β_q is $(c_q = 0 \vee c_q = 1 \vee c_2 = 2)$. We continue in a similar manner and after next $n - 2$ steps we obtain $\beta_q = (c = 0 \vee \cdots \vee c = n)$. The next iteration of this approach does not change the value of β_q.

As proposed above, if we replace the self-loop of q by its acceleration formula and apply the same approach, then we obtain the result after one iteration. That is, we take $\beta_q \wedge \exists k : (0 \leq k \leq n \wedge c'_q = c + k \wedge c'_q \leq n)$ and apply projection on c_q followed by the *unprime* function. The resulting formula is $(c = 0 \vee \cdots \vee c = n)$ and the new value of β_q is $(c = 0 \vee \cdots \vee c = n)$. Using this approach we save $n - 1$ iterations (note that n can be large in practice).

Algorithm 10 searches for a reachable final state with a satisfiable final condition in the product automaton $M_1 \times \overline{M_2}$. The algorithm is an application of breath-first search on $M_1 \times \overline{M_2}$ where the starting points are the initial states. Thus we add all initial states to the set $Worklist$ (Line 1). Before the main while loop, we initialize the formula $\beta_{(q,R)}$ for each product-state (q, R) (Line 2). If (q, R) is an initial state, i.e., $(q, R) \in Worklist$, then the possible values of counters in this state are given by the initial condition, this extraction is done by the function *init* (Lines 3, 4). For other states that are not initial, i.e., $(q, R) \notin Worklist$, the possible values are unknown, thus we set $\beta_q = \bot$ (Lines 5, 6).

Until $Worklist$ is empty, we take a product-state (q, R) from $Worklist$ (Lines 7, 8). The formula $\beta_{(q,R)}$ can be immediately updated if (q, R) has self-loops. The function *accelerate* updates $\beta_{(q,R)}$ according to the self-loops on (q, R) (Line 9). We show how $\beta_{(q,R)}$ is updated if the state contains only one self-loop. But what happened if (q, R) has several self-loops? Let $\varphi_1, \ldots, \varphi_n$ be self-loops on (q, R) in an arbitrary, but fixed, order. For each self-loop φ_i we know its acceleration formula ψ_i. Thus we also know how $\beta_{(q,R)}$ is updated. Suppose that $\chi_i = unprime(projection(\psi_i \wedge \beta(q, R)))$. If $[\![\chi_i]\!] \subseteq [\![\beta_{(q,R)}]\!]$, then we say that $\beta_{(q,R)}$ is not updated. The function *accelerate* works as follows: we update $\beta_{(q,R)}$ by processing the acceleration formula of the self-loops $\psi_1, \ldots, \psi_n$ repeatedly until last n acceleration formulae do not update $\beta_{(q,R)}$. Intuitively, if $\beta_{(q,R)}$ is updated, then the new values can be again changed by other self-loops (thus $\beta_{(q,R)}$ can be updated). But if n consecutive acceleration formulae of self-loops do not update $\beta_{(q,R)}$, then we know that any acceleration formula ψ_i also does not update $\beta_{(q,R)}$ because it also does not update in previous precessing of ψ_i. We note that *accelerate* does not update the formula $\beta_{(q,R)}$ if (q, R) has no self-loop.

After accelerating self-loops on (q, R) and updating $\beta_{(q,R)}$, we check whether (q, R) is a final state with a satisfiable final condition. If $IsSat(\beta_{(q,R)} \wedge F)$, then we return `FALSE` (Lines 10, 11). Otherwise we continue; we process all satisfiable outgoing transitions from (q, R) that are not self-loops (Lines 12, 13). If a transition is satisfiable, then we update the formula of the target state $\beta_{(q',R')}$ as described above (Line 14). If $[\![\psi]\!] \subseteq [\![\beta_{(q',R')}]\!]$, then we do not get any new information about the values of counters in (q', R'). So there is no reason to add (q', R') to $Worklist$ because if some outgoing transition from (q', R') was unsatisfiable before, then it will be also now. Otherwise, we add (q', R') to $Worklist$ and update the formula $\beta_{(q',R')}$ in order to reflect the new obtained information (Lines 16, 17).

Algorithm 10: Reachability of final states

Input : The product automaton $M_1 \times \overline{M_2} = (Q, C, I, F, \Delta)$
Output : `TRUE` if and only if $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$

1 $Worklist \leftarrow \{(q, R) \mid \mathtt{s} = (q, R) \text{ appears in } I\}$;
2 **foreach** $(q, R) \in Q$ **do**
3   **if** $(q, R) \in Worklist$ **then**
4         $\beta_{(q,R)} \leftarrow init(I)$;
5   **else**
6         $\beta_{(q,R)} \leftarrow \perp$;
7 **while** $Worklist \neq \emptyset$ **do**
8   pick and remove (q, R) from $Worklist$;
9     $accelerate(\beta_{(q,R)})$;
10   **if** $IsSat(\beta_{(q,R)} \wedge F)$ **then**
11   **return** `FALSE` ;
12   **foreach** $(q, R) \xrightarrow{\{\varphi\}} (q', R') \in \Delta$ such that $q \neq q'$ or $R \neq R'$ **do**
13   **if** $IsSat(\beta_{(q,R)} \wedge \varphi)$ **then**
14             $\psi \leftarrow unprime(projection(\beta_{(q,R)} \wedge \varphi))$;
15   **if** $[\![\psi]\!] \not\subseteq [\![\beta_{(q',R')}]\!]$ **then**
16                 $Worklist \leftarrow Worklist \cup \{(q', R')\}$;
17                 $\beta_{(q',R')} \leftarrow \beta_{(q',R')} \vee \psi$;
18 **return** `TRUE` ;

If $Worklist$ is empty, then it means that there is no reachable final state in $M_1 \times \overline{M_2}$. Thus we return `TRUE` (Line 18).

We note that states (q, R) can appear in $Worklist$ more than once. But always the semantics of the formula $\beta_{(q,R)}$ gets larger if the state (q, R) appears again in the $Worklist$ (Line 17). Since the number of possible configurations of the product automaton is finite (the product automaton is still an CA), the state (q, R) cannot be added to $Worklist$ infinite number of times. Therefore, the algorithm always terminates.

Finally, we point out that it is not necessary to build the whole product automaton and after that search for a reachable final state with a satisfiable final condition. The purpose of such a presentation was the simplification of our reasoning. In practice, we build the product automaton on-the-fly and if we encounter a reachable final state (with a satisfiable final condition), then we can stop. In this step, we have $\mathcal{L}(M_1) \cap \mathcal{L}(\overline{M_2}) \neq \emptyset$, or, equivalently, $\mathcal{L}(M_1) \not\subseteq \mathcal{L}(\overline{M_2})$. Eventually, we stop if we build the whole product automaton and we do not encounter on any final state (with a satisfiable final condition). In other words, we combine Algorithms 9 and 10 together.

Chapter 6

Experiments and Evaluation

In the previous chapter, we designed an algorithm solving the inclusion problem of MCAs. We implemented this algorithm in C++ and used the Z3 SMT solver [4] for the manipulation of formulae in our implementation (for more details see Section *Implementation* below). We recall that our algorithm can be used in a new method for testing inclusion problem of eREs—whether the language of the first eRE is included in the language of the second eRE. For transforming eREs into MCAs and determinizing of MCAs, we use Microsoft's Automata library [3]. Then we apply our implementation of the algorithm solving the inclusion problem of MCAs. In this chapter, we experimentally evaluate the performance of this method (denoted as **MCA** in the following) and compare it with the naive method (denoted as **NFA**), which is based on transforming eREs into NFAs (we used the implementation in Augeas Automata library [1]).

Nevertheless, the syntax of eREs is formally restricted by our definition (see Section 4.1), there is no problem transform any eREs appearing in practice into MCAs (all operators that are not in our definition can be simulated by only those operators used only in our definition) and still maintain the succinct representation. Since the implementation of the determinization algorithm of MCAs in [3] does not work properly for all cases, we restricted ourselves on the eREs where the counting is on the character class only in the forms $\sigma\{n\}$ or $\sigma\{n,\}$[1]. We note that our implementation should also work for the general case where the counting is of the form $\sigma\{m,n\}$.

For the experiments, we took a subset of 1014 eREs used in the experimental evaluation of determinization algorithm in [14]—namely, those used in network intrusion detection systems (Snort [20]: 260 eREs, Yang [24]: 102 eREs, Bro [22]: 403 eREs, HomeBrewed [9]: 36 eREs), the Microsoft's security leak scanning system (Industrial: 7 eREs), the Sagan log analysis engine (Sagan [6]: 1 eRE), and the patter matching rules from RegExLib (RegExLib [5]: 205 eREs). We note that each eRE contains at least one counting operator. In the following, this set of eREs is denoted by R and the same set without the eREs from Bro is denoted by R_{-Bro}.

In this chapter, we provide three experiments. In Section 6.1, we present an experiment where we randomly choose two eREs from R and check whether the language inclusion between them holds. In the experiment in Section 6.2, we again randomly choose two eREs but now we construct from them two other eREs in which the language inclusion holds. In the experiment in Section 6.3, we will take a look on „artificial“ pairs of eREs. These pairs are created by us motivated from the theoretical point of view or from practice. All

[1] Formally, the expression $\sigma\{n,\}$ is an abbreviation of $\sigma\{n\}\sigma^*$.

experiments were run on an *Intel Core i7-7500U* CPU@2.70GHz with 8GiB of RAM. Unless stated otherwise, the timeout in the experiments is 60 seconds.

Implementation

We are not going deep into implementation details. We only summarize the main points and identify the most inefficient part of our implementation. Overall, the implementation is a straightfroward combination of Algorithms 9 and 10 (see the last paragraph of Section 5.4), but instead of MCAs we used the so-called symbolic MCAs (see Example 2.4) with the algebra $\mathbf{BV}_{16}$ (see Example 2.2).

Recall that we use the Z3 SMT solver [4] for any manipulation with formulae in our implementation. For transforming input eREs to MCAs and determinization of MCAs we use Microsoft's Automata library [3]. Since Microsoft's Automata library provides an interface in .NET and our algorithm is implemented in C++, the automata generated by the library must be passed to our implementation via text interface. In fact, we use the DGML format for representing CAs [2]. For parsing DGML, we use PUGI XML [19] and the particular lines from DGML files we parse manually since we need to extract information such as names of counters or types of self-loops.

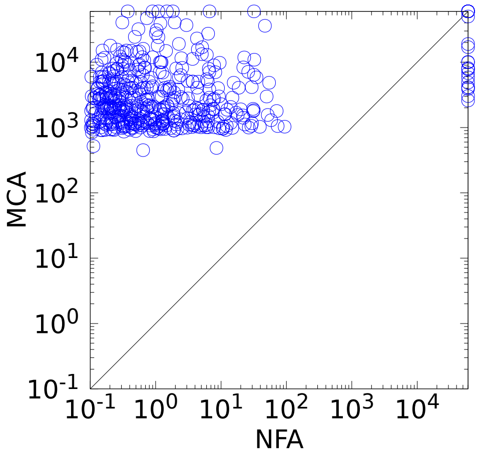
It is not efficient to save already constructed data structures to a file and from this file create the same data structures in C++. Thus there is a huge potential to improve the performance of our algorithm by integrating it into Microsoft's Automata library. Moreover, it is possible to modify the implementation of the determinization algorithm of MCAs in Microsoft's Automata library to determine the forms of self-loops already in the construction of DMCAs. The second possibility to improve the performance of our algorithm is to use the $\mathbf{BDD}_{16}$ algebra (see Example 2.1) instead of $\mathbf{BV}_{16}$. Furthermore, without an argument, our implementation has many opportunities for optimization.

Unfortunately, we are not able to implement the acceleration formula of the form III in Z3 using only linear integer logic. Thus if we encounter a self-loop containing a part of the form III, then we need to use the trivial acceleration of such a self-loop. Although, in practice (at least in our experiments), there are more self-loops of forms I and II than III, we think that this also increases the performance of the implementation.

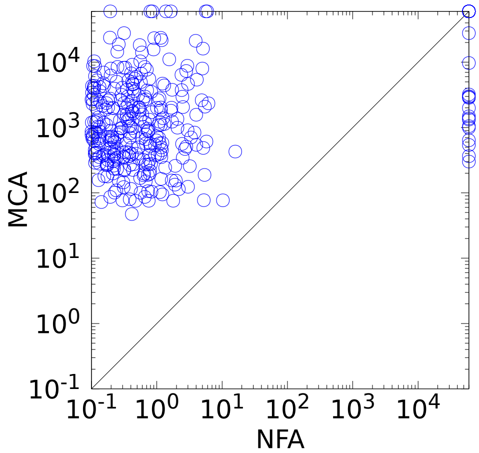
6.1 Random Pairs of Extended Regular Expressions

In the first experiment, we randomly chose 500 pairs (r_1, r_2) of eREs from R. In general, if a random pair is chosen, then the language inclusion of r_1 and r_2 does not hold, i.e., $\mathcal{L}(r_1) \nsubseteq \mathcal{L}(r_2)$. In fact, there were only 2 pairs among the 500 pairs in which the language inclusion holds. In Figure 6.1, we compare the running times of **MCA** and **NFA** on testing inclusion of the 500 chosen pairs of eREs. In the experiment were 29 **NFA** cases and 20 **MCA** cases where the algorithms timeouted (13 cases is the overlap). All cases that timeouted are plotted at the time 60 seconds.

Although **MCA** is less prone to explode than **NFA**, **NFA** outperforms **MCA** in every case when **NFA** finishes. Figure 6.1 (b), which gives the times without including the time needed to construct the automata, shows that the reason why **NFA** outperforms **MCA** is not only because of combining C++ and .NET via text interface. It generally holds that working with symbolic representation is slower than with explicit representation for easy cases. On average, each eRE used in this experiment has 1.64 counting operators with bound 111. The eREs with such a property are still not the hardest cases for us.



(a)



(b)

Figure 6.1: In (a) is a comparison of running times of **MCA** and **NFA** solving the inclusion problem of 500 random pairs of eREs. In (b) is the same comparison as in (a) where the time for constructing automata is subtracted. The time is given in milliseconds and the axes are logarithmic.

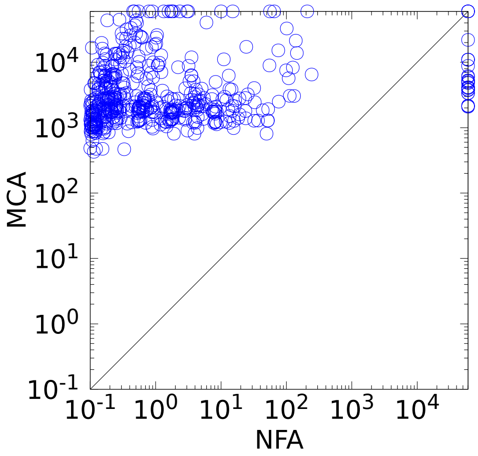
In Figure 6.1 (a), we also see that almost no experiment of **MCA** finishes within one second. This feature is not only in this experiment, but appears also in the next experiments. This is the cost of combining .NET and C++ via text interface.

6.2 Pairs of Extended Regular Expressions in which Inclusion Holds

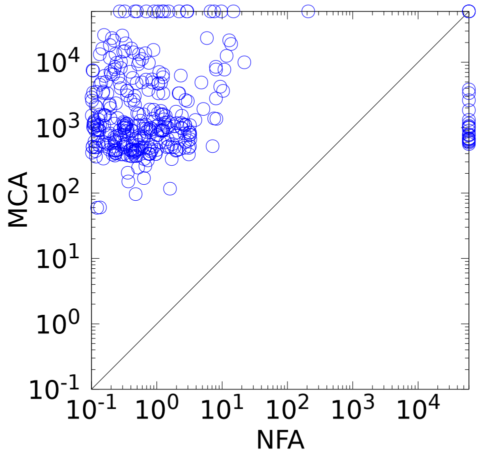
In the second experiment, we will take a look on pairs of eREs in which language inclusion holds. Since it is tough to find two distinct eREs from R in which language inclusion holds, we created our own pairs of eREs. Again, all cases that timeouted are plotted at time 60 seconds.

Figure 6.2 compares the running time of **MCA** and **NFA** on testing the 589 pairs of eREs of the form (r, r) where r is selected from R_{-Bro}. There is no big difference from the experiment of the 500 random pairs of eREs from the preceding section. The most interesting for us are the pairs that originate from Snort (see Section 6.3 for the structure of such eREs), because **NFA** timeouted in 28 cases and **MCA** only in 9 cases (the overlap is 4 cases). In all other pairs (different from Snort) **NFA** never timeouts but **MCA** timeouted in the next 13 cases (overall, **MCA** timeouted in 22 cases). Also in this experiments **NFA** outperforms **MCA** for all cases when **NFA** finished. One of the reason why **NFA** is faster is that the eREs are still relative easy—on average, each eRE from R contains 1.62 counting operators with the bound 112. Moreover, the product automaton constructed in **NFA** has 207 states on average (and it has 20 states on average in **MCA**).

In Figure 6.3, we compare the running times of **MCA** and **NFA** on testing inclusion of the 289 pairs of eREs where the second eRE from the pairs differs from the first eRE by the addition of the suffix .*. In other words, we testing the inclusion of the pairs $(r, r.^*)$ where r is randomly chosen from R. The addition does not significantly change the result of



(a)



(b)

Figure 6.2: In (a) is a comparison of running times of **MCA** and **NFA** solving the inclusion problem of 589 (r, r) pairs of eREs where r is chosen from R_{-Bro}. In (b) is the same comparison as in (a) where the time for constructing automata is subtracted. The time is given in milliseconds and the axes are logarithmic.

the graph, but it changed the number of timeouts. In particular, there were 10 **NFA** cases and 30 **MCA** cases where the algorithms timeouted (5 cases is the overlap.).

6.3 Artificial Pairs of Extended Regular Expressions

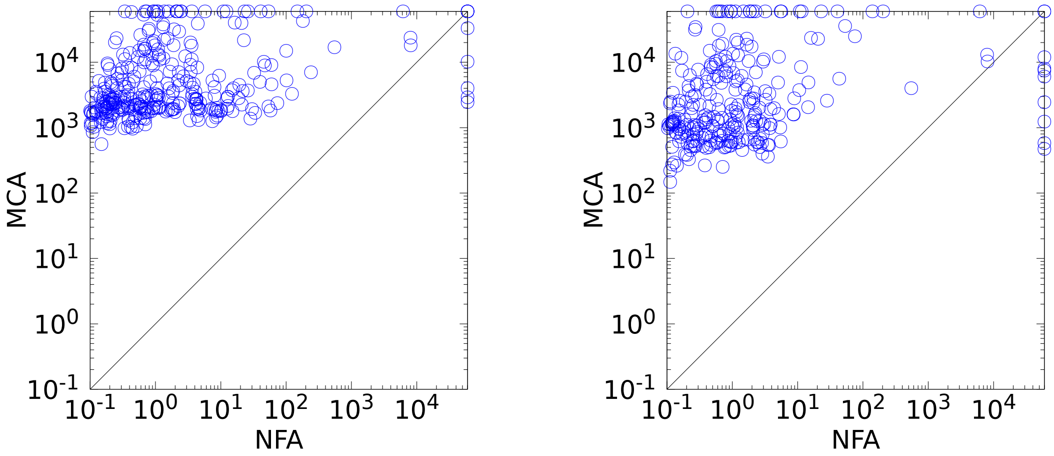
In this section, we consider „artificial“ eREs, which were created by us motived from the theoretical point of view or from the interesting eREs from the preceding sections. Namely, we examined pairs of eREs that originate from the following four pairs $(r_1, r_2), (s_1, s_2), (t_1, t_2)$, and (p_1, p_2) of eREs by varying `k` over $\mathbb{N}^+$:

$r_1 =$ `.*a.{k}`	$r_2 =$ `.*a.{k-1,}`,
$s_1 =$ `a{k}`	$s_2 =$ `a{k-1,}`,
$t_1 =$ `.*[aA][^\x0a]{k}`	$t_2 =$ `.*[aA][^\x0a]{k-1,}`,
$p_1 =$ `.*[aA][bB][cC][dD][^\x0a]{k}`	$p_2 =$ `.*[aA][bB][cC][dD][^\x0a]{k-1,}`.

For each pair $(r, r') \in \{(r_1, r_2), (s_1, s_2), (t_1, t_2), (p_1, p_2)\}$, we test $\mathcal{L}(r) \subseteq \mathcal{L}(r')$ for various value of `k` by using the algorithms **MCA** and **NFA** (note that the inclusion in the pair (r, r') holds for any positive `k`). Moreover, we justify the selection of such a pair. The timeout is 120 seconds except in the experiment with the pair (s_1, s_2) where the timeout is still 60 seconds as in the preceding sections.

(I) `.*a.{k}` and `.*a.{k-1,}`

Consider the first pair of eREs (r_1, r_2). The eRE r_1 is a well-known example where the smallest equivalent DFA has $2^{\mathtt{k}+1}$ states and the smallest equivalent DMCA has $\mathtt{k}+2$ states. Note that the smallest DFA and DMCA equivalent to r_2 have $2^{\mathtt{k}}$ and $\mathtt{k}+1$ states, respectively. Although the size of DFAs grow exponentially with `k`, the experiment shows that **NFA** outperform **MCA** for any `k`. In particular, the timeout in **MCA** already expired



(a)

(b)

Figure 6.3: In (a) is a comparison of running times of **MCA** and **NFA** solving the inclusion problem of 289 pairs $(r, r.^*)$ of eREs where r is randomly chosen from R. In (b) is the same comparison as in (a) where the time for constructing automata is subtracted. The time is given in milliseconds and the axes are logarithmic.

when k $= 5$ and in **NFA** when k $= 15$ (due to the small value of k we do not plot the graph in this experiment).

Each state (except the initial) in the DMCA equivalent to r_2 that is generated by the determinization algorithm (see Section 5.1) has two self-loops of the form II and III. Since we are not able to implement the acceleration formula of type III, the underlying formulae are too large. We suppose that implementation of the acceleration formula of type III helps to increase the performance of our implementation in this experiment.

(II) `a{k}` and `a{k-1,}`

The second pair (s_1, s_2) is interesting because the running time of **MCA** is constant regardless of the value k (we were only limited by the range of the integer in Z3, that is $2^{31} - 1$) as shown in Figure 6.4 where we limit the value of k by 50000. Note that **NFA** is better for all cases when k is less than 5000. The constant running time of **MCA** is given by the fact that the MCA for s_1 and the DMCA for s_2 have a constant number of states (in particular 2 and 3, respectively). Thus the formulae in the product automaton have the same structure and differ only by the value of k.

(III) `.*[aA][^\x0a]{k}` and `.*[aA][^\x0a]{k-1,}`
(IV) `.*[aA][bB][cC][dD][^\x0a]{k}` and `.*[aA][bB][cC][dD][^\x0a]{k-1,}`

The fourth pair (p_1, p_2) is a representation of eREs from Snort where **MCA** significantly outperforms **NFA** in the experiments in Section 6.2. The general format is the following: eRE starts with .*, followed by a sequence of symbols or character classes without counting, and finished by a character class with counting `[^\x0a]{k}`. The biggest value of k found was 1024. One real example from Snort is the eRE

```
.*[pP][aA][sS][sS] [^\x0a]{100}.
```

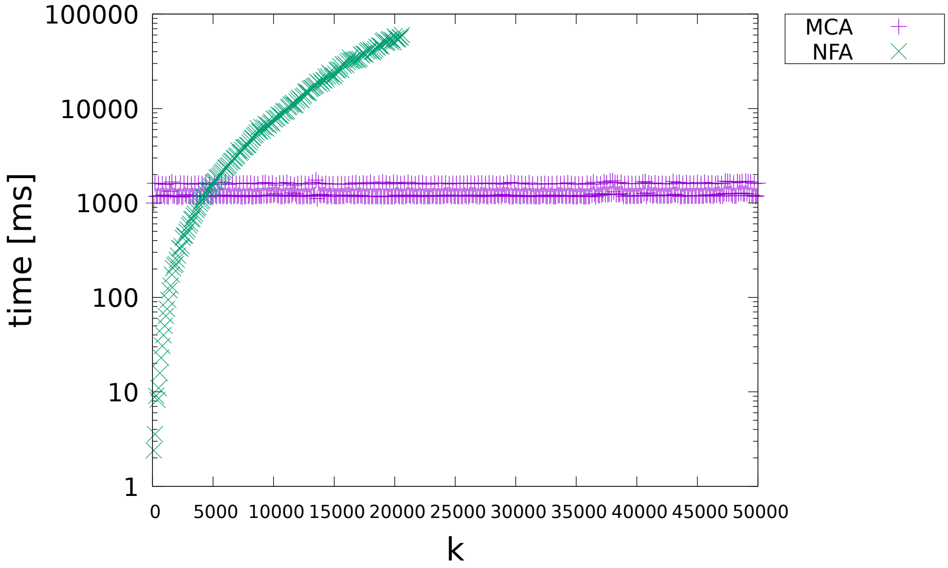


Figure 6.4: The running times of **MCA** and **NFA** solving the inclusion problem of `a{k}` and `a{k-1,}` where `k` starts from 100 and is incremented by 100 until the algorithm timeouts or `k` is equal to 50000. The vertical axis is logarithmic.

The third pair (t_1, t_2) is a special case of this general format.

In Figure 6.5 is plotted how running times of **MCA** and **NFA** depends on `k` in solving the inclusion problem of t_1 and t_2. We see that **NFA** timeouts for $\texttt{k} = 5$ while **MCA** timeouts for $\texttt{k} = 1199$. In Figure 6.6 is plotted how running times of **MCA** and **NFA** depends on `k` in solving the inclusion problem of p_1 and p_2. Now **NFA** timeouts for $\texttt{k} = 27$ and **MCA** timeouts for $\texttt{k} = 563$. Note that the value of `k` when **NFA** timetous increases but the value decreases when **MCA** timeouts. By observation this trend continues—the difference of performance between **MCA** and **NFA** decreases if the number of symbols between .* and `[^\x0a]{k}` are growing.

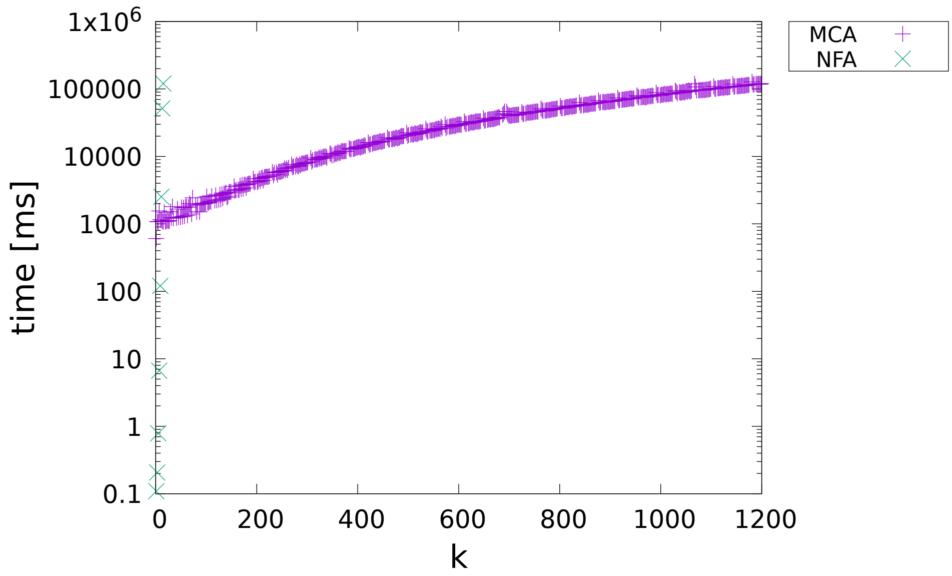


Figure 6.5: The running times of **MCA** and **NFA** solving the inclusion problem of `.*[aA][^\x0a]{k}` and `.*[aA][^\x0a]{k-1,}` where `k` starts from 1 and is incremented by 2 until the algorithm timeouts. The vertical axis is logarithmic.

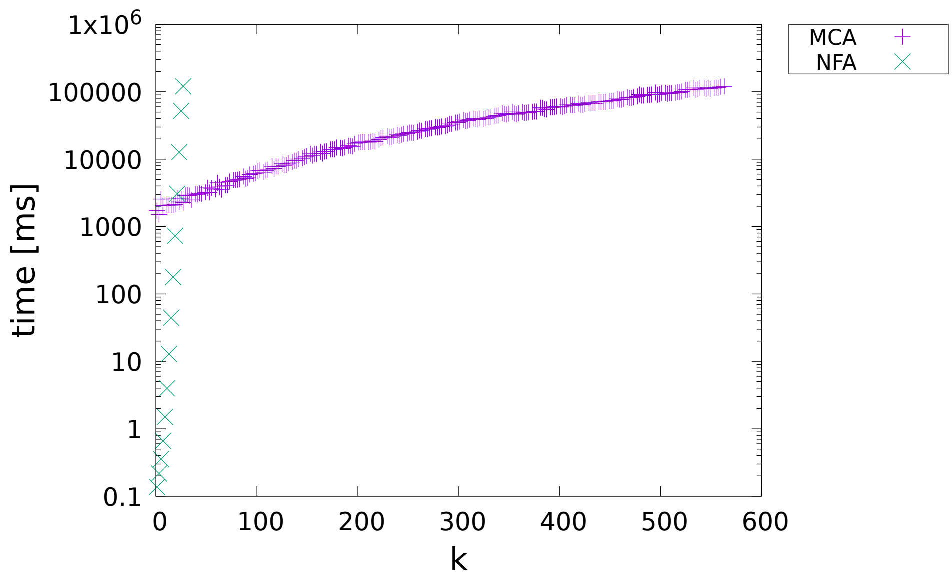


Figure 6.6: The running times of **MCA** and **NFA** solving the inclusion problem of `.*[aA][bB][cC][dD][^\x0a]{k}` and `.*[aA][bB][cC][dD][^\x0a]{k-1,}` where `k` starts from 1 and is incremented by 2 until the algorithm timeouts. The vertical axis is logarithmic.

Chapter 7

Conclusion

In this thesis, we efficiently solved the emptiness and inclusion problems of MCAs by imitating the solution to the inclusion problem of NFAs. To be able to imitate such a solution we had to find answers to the following: how to complement an MCA, how to construct the product automaton of an MCA and the complement of an MCA, and how to determine unreachable states in the product automaton. We also develop an intuition about why the emptiness and inclusion problems of general CAs require transformation to the NFAs. Moreover, we extended the class of MCAs to two larger subclasses of CAs. As we provided in examples, these subclasses are capable of representing more complex extended regular expression—we are not limited only to the counting on character classes, but for example, we can have counting on sequences of symbols. For these two subclasses of CAs, we gave an efficient solution to the emptiness problem.

We combined our implementation of the algorithm solving the inclusion problem of MCAs with the existing implementation of transforming eREs into MCAs and determinization of MCAs provided in Microsoft's Automata library [3]. This combination gives a new method for testing inclusion of eREs. We experimentally evaluated such a method on eREs from a wide range of applications and compared it with the naive method, which is based on transforming eREs into NFAs. Despite our implementation of the inclusion problem of MCAs is not optimized and we are not able to implement one acceleration formula of the self-loops in the determinized MCAs using only linear integer logic in the Z3 SMT solver [4], the experiments show that the method based on MCAs is less prone to explode. This holds especially, if the MCAs arise from the eREs that are used in the Snort [20] network intrusion detection system. Moreover, the method based of MCAs significantly outperforms the naive method in eREs where the counting operators have large bounds. On other hand, in the easy cases from practice (where the eREs have 1.6 counting operators with the bound 110 on average), the naive method is faster because it uses explicit representation.

Besides the designed algorithms and subclasses of CAs, we thoroughly investigated the structure of determinized MCAs that are the result of the application of determinization algorithm of MCAs in [14, Section 4.2]. This knowledge was used to accelerate the solution of the inclusion problem of MCAs, but the same approach can be used in minimization of determinized MCAs (for the purpose of removing unreachable states). Moreover, we think that the existing determinization algorithm of MCAs can be modified to generate only reachable states by using a similar method that we use in the acceleration of the inclusion problem.

Although we found all acceleration formulae of the self-loops in determinized MCAs, we were not able to implement one acceleration formula in the Z3 SMT Solver using only linear integer logic. We hope that the existence of such an implementation increases the performance of our implementation of the inclusion problem of MCAs. It would be great to try a more efficient algebra for the representation of symbol guards in (determinized) MCAs than the algebra used in our implementation since the used algebra is not the most efficient one. We also believe that the approach in the language inclusion of MCAs can be also applicable to a larger class than MCAs. Moreover, we see an opportunity to integrate our algorithm inside to Microsoft's Automata Library, which already provides an algorithm for determinizing MCAs.

Last but not least, there exist several solutions for the inclusion problem of NFAs. The simplest one (but not the most efficient) is based on the *subset construction*. The more complex approaches to solve the inclusion problem of NFAs use simulation (see Section 3.3) or antichains (when simulation is the identity relation in Section 3.3). The simulation or antichains are then used to prune out the unnecessary search path in searching for a final state. Our solution to the inclusion problem of MCAs can be categorized as the solution based on the subset construction. We see here a possibility to extend our solution to use the antichains approach since it does not need to have a special algorithm for computing simulation on MCAs.

Bibliography

[1] *Augeas:* `libfa` *Finite Automata Library* [online; 11. 5. 2020]. Available at: http://augeas.net/libfa/.

[2] *Directed Graph Markup Language (DGML)* [online]. Available at: http://schemas.microsoft.com/vs/2009/dgml/.

[3] *Microsoft Automata library: Automata and Transducer Library for .NET* [online; 25. 2. 2020]. Available at: https://github.com/AutomataDotNet/Automata.

[4] *Microsoft Research: Z3 Theorem Prover* [online; 6.3.2020]. Available at: https://github.com/Z3Prover/z3.

[5] *RegExLib.com: The Internet's First Regular Expression Library* [online; 12.5.2020]. Available at: https://regexlib.com/.

[6] *The Sagan team: The Sagan Log Analysis Engine* [online; 12.5.2020]. Available at: https://quadrantsec.com/sagan_log_analysis_engine/.

[7] Abdulla, P. A., Chen, Y.-F., Holík, L., Mayr, R. and Vojnar, T. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 158–174. TACAS'10. ISBN 3642120016. Available at: https://doi.org/10.1007/978-3-642-12002-2_14.

[8] Andersen, H. R. *An Introduction to Binary Decision Diagrams*. 1999.

[9] Češka, M., Havlena, V., Holík, L., Lengál, O. and Vojnar, T. Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018, p. 155–175. ISBN 978-3-319-89963-3.

[10] D'Antoni, L. and Veanes, M. Minimization of Symbolic Automata. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. January 2014, vol. 49, no. 1, p. 541–553. ISSN 0362-1340. Available at: https://doi.org/10.1145/2578855.2535849.

[11] Esparza, J. *Automata Theory: An Algorithmic Approach* [online; 18.12.2019]. Available at: https://www7.in.tum.de/~esparza/automatanotes.html.

[12] Gorrieri, R. *Process Algebras for Petri Nets: The Alphabetization of Distributed Systems*. 1st ed. Springer Publishing Company, Incorporated, 2017. 15-34 p. ISBN 3319555588.

[13] Holík, L., Lengál, O., Síč, J., Veanes, M. and Vojnar, T. Simulation Algorithms for Symbolic Automata. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Springer, 2018, vol. 11138, p. 109–125. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-030-01090-4_7.

[14] Holík, L., Lengál, O., Saarikivi, O., Turoňová, L., Veanes, M. and Vojnar, T. Succinct Determinisation of Counting Automata via Sphere Construction. In: *In Proc. of 17th Asian Symposium on Programming Languages and Systems - APLAS'19*. Springer Verlag, 2019, no. 11893, p. 468–489. ISSN 0302-9743. Available at: https://www.fit.vut.cz/research/publication/12077.

[15] Hooimeijer, P. and Veanes, M. An Evaluation of Automata Algorithms for String Analysis. In: *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 248–262. ISBN 978-3-642-18275-4.

[16] Hopcroft, J. E. *An n log n Algorithm for Minimizing States in a Finite Automaton*. Stanford, CA, USA, 1971.

[17] Hovland, D. Regular Expressions with Numerical Constraints and Automata with Counters. In: *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*. Berlin, Heidelberg: Springer-Verlag, 2009, p. 231–245. ICTAC '09. ISBN 9783642034657. Available at: https://doi.org/10.1007/978-3-642-03466-4_15.

[18] Ilie, L., Navarro, G. and Yu, S. On NFA reductions. In: *Lecture Notes in Computer Science*. Springer, January 2004, vol. 3113, p. 112–124.

[19] Kapoulkine, A. *pugixml* [online; 13.3.2020]. Available at: https://pugixml.org/.

[20] Roesch, M. *Snort: A Network Intrusion Detection and Prevention System* [online; 12.5.2020]. Available at: https://www.snort.org/.

[21] Smith, R., Estan, C., Jha, S. and Siahaan, I. Fast Signature Matching Using Extended Finite Automaton (XFA). In: *Proceedings of the 4th International Conference on Information Systems Security*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 158–172. ICISS '08. ISBN 9783540898610. Available at: https://doi.org/10.1007/978-3-540-89862-7_15.

[22] Sommer, R. et al. *The Bro Network Security Monitor* [online; 12.5.2020]. Available at: http://www.bro.org.

[23] Veanes, M. Applications of Symbolic Finite Automata. In: *Lecture Notes in Computer Science*. Springer, July 2013, vol. 7982, p. 16–23.

[24] Yang, L., Karim, R., Ganapathy, V. and Smith, R. Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In: *Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 58–78. ISBN 978-3-642-15512-3.