BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

BEYOND REGISTER AUTOMATA: PUSHING THE BORDER OF DECIDABILITY

ZA REGISTROVÝMI AUTOMATY: POSOUVÁNÍ HRANIC ROZHODNUTELNOSTI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **SABÍNA GULČÍKOVÁ**
AUTOR PRÁCE

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

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Gulčíková Sabína**
Programme: Information Technology
Title: **Beyond Register Automata: Pushing the Border of Decidability**
Category: Theoretical Computer Science

Assignment:

1. Get acquainted with the theory of *register automata* (also called *finite-memory automata*).
2. Select an extension of the basic register automata model, such as register automata over a finite domain, register automata over a structured infinite domain, or register automata that can hold structured values in their registers (such as a set of elements of the input domain).
3. Based on the selected extension, either:
    1. develop the necessary formal theory for the given model, focusing on Boolean closure properties, the power of nondeterminism/alternation, decidability properties, size reduction, and the relation to other register automata models and fragments of logic, or
    2. develop efficient algorithms for working with the given model, focusing on efficient handling of operations such as inclusion testing, reduction, etc.
4. In the case the second point of (3) was chosen, implement the developed algorithms and compare them experimentally with algorithms that work on finite automata (in the case an extension to finite domains was chosen) or other algorithms for handling register automata, for example on benchmarks from problems in the theory of strings.
5. Discuss the obtained results and possible further extensions.

Recommended literature:

- Michael Kaminski, Nissim Francez: Finite-Memory Automata. Theor. Comput. Sci. 134(2): 329-363 (1994)
- Stéphane Demri, Ranko Lazic: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. 10(3): 16:1-16:30 (2009)
- Diego Figueira: Alternating register automata on finite words and trees. Log. Methods Comput. Sci. 8(1) (2012)

Requirements for the first semester:

- Items 1 and 2.

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, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

Register automaton (RA) operating over infinite alphabet is one of the great tools for pattern matching with backreferences, runtime verification, or modelling of parallel computation. In case of pattern matching with backreferences, the state-of-the-art matchers make use of backtracking algorithms, whose application causes significant slowdown in case of nondeterministic regular expressions.

Since RAs cannot always be determinised, it is an unsuitable model for solution to problems related to inefficient usage of backtracking algorithms. On the other hand, the RA's quality of being equipped by a finite memory serves as a good basis for storing the so-called capture groups used in pattern matching with backreferences.

In this work, a formal model called register set automaton (RsA) is proposed. A large class of RAs can be transformed into this deterministic model, which, among other things, allows for fast pattern matching with backreferences. We explore RsA's properties including decidability of emptiness testing, determinisability, closure under Boolean operations and we compare it to other register models in context of their expressive power.

Abstrakt

Registrový automat (RA) pracujúci nad nekonečnou abecedou je jedným z nástrojov pre pattern matching s backreferenciami, dynamickú verifikáciu, alebo modelovanie paralelných výpočtov. Súčasné riešenia v aplikáciách pattern matchingu používajú backtrackingové algoritmy v prípade nedeterministických regulárnych výrazov.

Nemožnosť determinizovať registrový automat spôsobuje, že nie je vhodným formálnym modelom pre riešenie problémov spojených s neefektívnymi aplikáciami backtrackingových algoritmov. Na druhej strane, vybavenosť konečnou pamäťou slúži ako vhodná báza pre ukladanie takzvaných capture groups použitých v takejto aplikácii.

Táto práca sa zaoberá predstavením formálneho modelu registrovo množinového automatu. Veľká trieda registrových automatov môže byť transformovaná do tohto deterministického modelu, ktorý okrem iného, dovoľuje vykonávať rýchly pattern matching s backreferenciami. Definované sú vlastnosti zahŕňajúce rozhodnutelnosť testu prázdnosti, determinizovateľnosť, uzavretosť voči Booleovským operáciám. Zároveň tento model porovnávame voči iným registrovým modelom z hľadiska ich vyjadrovacej sily.

Keywords

finite memory automata, register automata, infinite alphabet, regular expressions, regular expressions with backreferences, language inclusion of register automata

Kľúčové slová

automaty s konečnou pamäťou, registrové automaty, nekonečná abeceda, regulárne výrazy, regulárne výrazy s backreferenciami, jazyková inklúzia registrových automatov

Reference

GULČÍKOVÁ, Sabína. *Beyond Register Automata: Pushing the Border of Decidability.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšírený abstrakt

Motivácia. Registrový automat (RA) je automatový model pracujúci nad nekonečnou abecedou. Modely pracujúce nad nekonečnou abecedou sú využité v rôznych praktických aplikáciách, akými je napríklad dynamická verifikácia, pattern matching s backreferenciami, alebo modelovanie sieťových protokolov. Model registrového automatu bol predstavený ako rozšírenie konečného automatu. Toto rozšírenie je popri klasickej štruktúre pozostávajúcej zo stavov a prechodov vybavené konečnou pamäťou vo forme konečnej množiny registrov. V každom registri možno uchovávať jednu hodnotu videnú na vstupnej páske, a neskôr voči tejto hodnote porovnávať obsahy iných registrov, či aktuálnu hodnotu na páske. Tento model je uzavretý voči prieniku a zjednoteniu, nie však voči komplementu. Test prázdnosti jazyka registrového automatu je rozhodnuteľný. Okrem neuzavretosti voči komplementu je registrový automat nie vždy determinizovateľný.

Keďže registrový automat nemožno vždy determinizovať, jeho použitie v praktických aplikáciách má nie vždy efektívny výsledok. Príkladom takejto aplikácie je provnávanie zadaného vstupu a rozšírených regulárnych výrazov s backreferenciami. Táto funkcia je často vykonávaná na rôznych webových stránkach, či počas spracovávania textu pomocou `grep` a `sed` nástrojov. Pre príklad uvážme nasledujúci regulárny výraz:

$$R_{\backslash 3\backslash 2\backslash 1} = \texttt{/(.).*;.*(.).*;.*(.).*\3\2\1/}.$$

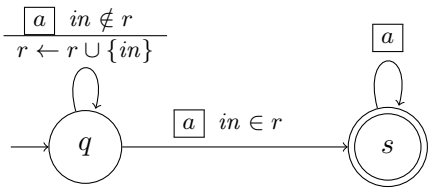
Pri použití nástroja PCRE2, ktorý predstavuje súčasné riešenia, trvá 10,416 krokov, kým nástroj vyhodnotí, že nasledovný, 42-znakov dlhý, náhodne vygenerovaný reťazec tomuto regulárnemu výrazu nevyhovuje:

`"ah;jk2367ash;la5akv45lwkjb9f.dj5fqkbxsfyrf"`.

Toto enormné spomalenie je spôsobené tzv. katastrofickým backtrackingom. Keďže PCRE2 nie je založený na deterministickom modeli, pri kontrole, či zadaný vstup vyhovuje konkrétnemu regulárnemu výrazu, musí nad daným slovom nedeterministicky vykonať mnoho behov, kým nenarazí na jeden prijímajúci. Takéto spomalenie vedie k náchylnosti systému, ktorý danú techniku kontroly využíva, na zlyhania spôsobené ReDoS útokmi. Okrem vopred zmienenej aplikácie je použitie deterministického registrového modelu žiadúce pre kontrolu jazykovej inklúzie registrových automatov, čo je vo všeobecnosti nerozhodnuteľný problém.

Cieľom tejto práce je predstaviť model, ktorý nám umožní transformovať veľkú triedu registrových automatov do ich deterministickej podoby. Pre daný problém identifikujeme jeho rôzne vlastnosti, vrátane uzavretosti voči Booleovským operáciám, rozhodnuteľnosti a konkrétnej zložitosti niektorých rozhodovacích problémov. Práca je tiež zameraná na určenie vyjadrovacej sily predstaveného modelu v porovnaní s inými registrovými modelmi.

Navrhnutý model. Navrhnutý model je založený na rozšírení registrového automatu. Takzvaný *registrovo množinový automat* (RsA) je štruktúrou podobný RA, s výnimkou možného obsahu registrov. V prípade RsA v každom registri možno ukladať *množinu* hodnôt videných na vstupnej páske. Nasledovne je možné testovať príslušnosť, či neprís-lušnosť aktuálnej hodnoty na vstupe do konkrétneho registra. Do registrov možno pridávať akutálnu hodnotu zo vstupu, alebo do nich zjednocovať niekoľko obsahov iných registrov. Obrázok 1 obsahuje ukážku registrovo množinového automatu, ktorý prijíma reťazce, v ktorých sa akýkoľvek symbol vyskytuje aspoň dvakrát. Vyobrazený automat je deterministický. Je nutné poznamenať, že tento jazyk nie je reprezentovateľný deterministickým registrovým automatom.



Obrázok 1: Príklad deterministického registrovo množinového automatu.

Vlastnosti RsA**.** Uzáverové vlastnosti registrovo množinového automatu sú rovnaké ako v prípade registrových automatov. RsA je uzavretý na zjednotenie a prienik, nie je uzavretý na komplement. Keďže registrovo množinové automaty generalizujú nedeterministické registrové automaty, problémy univerzality, ekvivalencie a jazykovej inklúzie sú pre tento model nerozhodnuteľné.

Jednou z dôležitých vlastností registrových automatov je, že test prázdnosti ich jazyka je rozhodnuteľný. Presnú zložitosť tohto rozhodovacieho problému možno určiť pomocou obojsmernej redukcie na test pokryteľnosti v transferových Petriho sieťach, čo je známy, $\mathbf{F}_\omega$-úplný problém. Transferové Petriho siete sú rozšírením klasických Petriho sietí o tzv. broadcastové hrany, ktoré nám umožňujú prénašať všetky tokeny zo zdrojového miesta prechodu, do cieľového miesta prechodu. Model transferových Petriho sietí úzko súvisí s broadcastovými protokolmi.

Determinizácia NRA **do** DRsA**.** RsA majú nasledujúcu vlastnosť: veľká trieda registrových automatov môže byť determinizovaná do deterministických registrovo množinových automatov. Navrhnutý algoritmus pre túto transformáciu je založený na klasickom algoritme determinizácie vykonávajúcom podmnožinovú konštrukciu, s pridaným spracovaním hodnôt uložených v registri. Spracovanie zahŕňa detekciu nadaproximácie, alebo zmeny semantiky výsledného jazyka automatu, ku ktorej môže dôjsť v rôznych prípadoch. Navrhnutý algoritmus je, okrem iného, úplný pre triedu $\mathrm{NRA}_1^=$, čo je trieda nedeterministických registrových automatov s jedným registrom a výhradnými testami rovnosti na prechodoch.

Vyjadrovacia sila. Súčasťou tejto práce bolo porovnanie predstaveného modelu s inými modelmi nad nekonečnými abecedami, a klasifikácia jeho vyjadrovacej sily voči iným, registrovým modelom. Predstavený model generalizuje registrové automaty, teda vyjadrovacia sila RsA je väčšia než vyjadrovacia sila RA. Tiež bolo na základe identifikácie niekoľkých jazykov zistené, že registrovo množinové automaty sú neporovnateľné s modelom alternujúceho registrového automatu, ani s jeho rozšírením o *guess* a *spread* operátory. Model RsA je tiež neporovnateľný s tzv. *pebble* automatmi.

Možné rozšírenia. V našej práci sme skúmali niekoľko rôznych rozšírení RsA. Skúmané rozšírenia spočívali v úprave prechodovej relácie, pričom sme na každom prechode umožnili vykonávať viacero operácii nad registrami. Aj malé rozšírenia prechodovej relácie vedú k nerozhodnuteľnosti. Pre príklad, keď umožníme testovanie rovnosti registrov, v prípade testu prázdnosti je možné ukázať redukovateľnosť z problému dosiahnuteľnosti v Petriho sieťach s inhibičnými hranami, čo je známy, nerozhodnuteľný problém. Rovnaký dôsledok platí aj pre rozšírenia o možnosť odoberania hodnôt z registrov, či o testovanie prázdnosti registra.

Zhrnutie a budúca práca. V tejto práci sme predstavili registrovo množinové automaty, triedu automatov pracujúcich nad dátovými slovami, ktoré predstavujú vhodnú formálnu bázu pre efektívny pattern matching podtriedy rozšírených regulárnych výrazov s backreferenciami. Navrhnutý model oplýva niekoľkými zaujímavými vlastnosťami. Je uzavretý voči prieninku a zjednoteniu, veľká trieda registrových automatov môže byť determinizovaná do registrovo množinových automatov, a test prázdnosti jazyka je pre tento model rozhodnuteľný. Súčasťou budúcej práce je experimentovanie s inými štruktúrami registrov, upravenie algoritmu determinizácie pre rozšírenie triedy registrových automatov determinizovateľných do registrovo množinových automatov, idenitifkácia korešpondujúceho fragmentu logiky pre deterministické RsA, a návrh algoritmov pre efektívne testovanie jazykovej inklúzie.

Beyond Register Automata: Pushing the Border of Decidability

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.

.....
Sabína Gulčíková
May 17, 2022

Acknowledgements

I would like to express my deepest gratitude to the supervisor of this thesis, Ing. Ondřej Lengál, Ph.D. for his immense patience, tolerance, his guidance, and the abundance of knowledge he was willing to share. I sincerely appreciate his pearls of wisdom, and useful remarks on the quality of this work. Thank You for making me replace each *which* with *that*, and vice versa! I am also grateful to Tibi, for his tremendous support, advice, and for hiding books from myself so that I would not procrastinate, and to Anička for keeping me company in all of my endeavors. Special thanks to my parents, for their endless support and words of encouragement.

Contents

Chapter 1

Introduction

Register automaton (RA) is an automaton model operating over an infinite alphabet. Such models have found applications in many fields of computer science including modelling network protocols [5], pattern matching with backreferences [34], runtime verification [15], or even testing satisfiability in the SMT theory of strings [6].

RA, also known as finite memory automaton, was first introduced in [22] as an extension of finite automaton equipped with a finite set of registers, each of them able to store a single value copied from the input tape. It was introduced with the goal of preserving closure under Boolean operations. The authors succeeded in doing so, with the exception of closure under complementation. This property could only be accomplished by constraining to a more restricted model. The emptiness problem, however, remained decidable. In addition, RAs not only cannot be complemented, they also cannot be determinised.

The RA's non-determinisability causes it to be an unsuitable model for some practical applications. One of them is matching regexes with back-references, which is done ubiquitously in validating user input on web pages, processing text using the `grep` and `sed` tools, transforming XML documents, or detecting network incidents [32, 4]. Consider, for instance, the (extended) regular expression (regex)

$$R_{\backslash 3\backslash 2\backslash 1} = \texttt{/(.).*;.*(.).*;.*(.).*\3\2\1/},$$

which matches strings of the form

$$\ldots\texttt{a}\ldots;\ldots\texttt{b}\ldots;\ldots\texttt{c}\ldots\texttt{cba}\ldots$$

for any symbols `a`,`b`,`c`, and any number of arbitrary symbols (except "`;`") at the positions of "`.`".

Using the state-of-the-art PCRE2 regex matcher, it takes 10,416 steps before reporting no match in the randomly generated 42-character-long string

```
"ah;jk2367ash;la5akv45lwkjb9f.dj5fqkbxsfyrf".
```

If we were to remove the delimiters (semicolons in $R_{\backslash 3\backslash 2\backslash 1}$), the process would take 179,372 steps in the same text. Ideally, this process should only take 42 steps, one for each character in the string. This slowdown is caused by the so-called *catastrophic backtracking*. The PCRE2 matcher is based on backtracking, and since the regex is nondeterministic, the backtracking algorithm needs to try all possibilities of placing the three capture groups before concluding that there is no match. Such scenario causes the systems making use of

pattern matching with backreferences to perform poorly. In the worst case, this leads to an undesirable behavior, with systems being prone to attacks such as *regular expression denial of service* (ReDoS) [2]. For instance, a 2016 StackOverflow outage was caused by a ReDoS attack [42]. In particular, the attack was conducted by creating a malformed post, containing several thousands of whitespaces, which in combination with a simple regex for removing the whitepaces caused a high consumption of CPU. This led to a collapse of StackOverflow's servers, making the website unavailable for more than one hour.

In addition to prevention of catastrophic backtracking, the existence of a deterministic register automaton model would be useful in language inclusion checking of register automata, which is in general an undecidable problem. Even though it is impossible to determinise a significant class of register automata, the presence of its finite memory lays out a convenient basis for storing of capture groups in pattern matching with backreferences. The main goal of this thesis is to introduce a formal model called register set automaton (RsA), which is based on extending the existing model of register automata, such that a deterministic subclass of RsAs can capture a large fragment of nondeterministic register automata (NRAs).

This work is directed towards the development of formal theory for RsAs, focusing on Boolean closure properties, the power of nondeterminism, and their position in the landscape of register automata models, in the context of their expressive power. In addition, we explore the decidability of problems such as emptiness testing in case of RsAs and their extensions. This thesis is organized in the following way. Chapter 2 contains the definitions of basic notions used throughout this work. In this chapter, automata theory is explained to the extent necessary for understanding the relation of the newly proposed model to the already existing ones. In Chapter 3, we introduce the proposed model and discuss its properties, including the closure under Boolean operations, decidability of some decision problems, and the power of nondeterminism. Chapter 4 contains the involved proof of decidability of emptiness problem for the introduced model together with the proof of its particular computational complexity. In Chapter 5, we discuss the determinisation of register automata into our proposed model, give a semi-algorithm for determinisation, and observe for which subclass of register automata it is complete. Chapter 6 is devoted to positioning the proposed model in the landscape of other register automata models, in the context of its expressive power. Finally, in Chapter 7, we discuss some possible extensions of the proposed model, and we give the main results for each of these models, in terms of decidability of some decision problems.

Chapter 2

Preliminaries

This chapter contains the definitions of concepts that are necessary for understanding the problems dealt with throughout this work. Section 2.1 contains the characterizations of notions that make up the formal basis of the developed theoretical model. Section 2.2 offers an overview of phenomena important for understanding the proposed extension of register automata. In addition, a brief definition of other related formal models follows. Register automata are compared to these models in terms of expressive power in the latter chapters. Section 2.1.1 contains definitions of some example languages, which are used throughout this work. Section 2.3 provides an introduction to regular expressions. Manipulation with their extensions is one of the motivations for this thesis. It is important to remark that throughout this work, the terms *register automaton* and *finite memory automaton* are used interchangeably.

2.1 Basic Notions

We use $\mathbb{N}$ to denote the set of natural numbers without 0, $\mathbb{N}_0$ to denote $\mathbb{N} \cup \{0\}$, and $[n]$ for $n \in \mathbb{N}_0$ to denote the set $\{1, \ldots, n\}$ (we note that $[0] = \emptyset$). Moreover, $f \colon X \rightharpoonup Y$ is used to denote a partial function f from X to Y. If the value of f for $x \in X$ is undefined, we write $f(x) = \perp$. In addition to denoting concatenation, we sometimes use "$\cdot$" to denote an *ellipsis*, i.e., a value that can be ignored. Next, we give some definitions for the terms used in the theory of automata.

Definition 2.1.1. *An* alphabet *is a finite, non-empty set of elements called* characters, letters, *or* symbols. *By* Σ^**, we denote a set of all words over alphabet* Σ*, and by* Σ^+*, the set of all non-empty words over alphabet* Σ*. A* data domain *is a countably infinite, non-empty set of elements called* data values. *A* word *over an alphabet* Σ *is a sequence of characters* $a_1 a_2 \ldots a_n$*, such that for all* $1 \leq i \leq n$*, it holds that* $a_i \in \Sigma$*.*

Example 2.1.1. *In example, let* $\Sigma = \{a\}$*. Then:*

- $\{a\}^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$*, and*
- $\{a\}^+ = \{a, aa, aaa, aaaa, \ldots\} = \{a\}^* \setminus \{\epsilon\}$.

Let us fix a finite nonempty *alphabet* Σ and an infinite *data domain* $\mathbb{D}$.

Definition 2.1.2. *A* data word *of* length n *is a function* $w \colon [n] \to (\Sigma \times \mathbb{D})$.

We use $|w| = n$ to denote the length of a word w. The *empty word* of length 0 is denoted as ϵ, which holds for both words and data words. The notations $\Sigma[w]$ and $\mathbb{D}[w]$ are used to denote the *projection* of data word w onto the respective domain.

Example 2.1.2. *Let $w = \langle a, 1\rangle\langle b, 2\rangle\langle b, 3\rangle$. Then, its Σ-projection and $\mathbb{D}$-projection are defined as $\Sigma[w] = abc$ and $\mathbb{D}[w] = 123$, respectively.*

Additionally, given $a \in \Sigma$, we use $a[w_i]$ as an abbreviation for $\Sigma[w_i] = a$. The number of occurrences of a symbol a in a word w is denoted by $\#_a(w)$.

Intuitively, the Σ portion of the data word defines the *context* of the data value it is paired with. As an example, consider a system with unbounded number of processes, each of which can be in one of two states, either *busy* (b), or *waiting* (w). When working with traces of such a system, each pair of the data word represents the identification of a given process, and its current state. An example of such data word is $\langle w, 8\rangle\langle b, 42\rangle\langle b, 99\rangle\langle w, 12\rangle$ [26].

In the following, we introduce some operations on words. They can be easily extended to data words.

Definition 2.1.3. *Let u and v be two words over alphabet Σ. A binary operation of concatenation, denoted by "$\cdot$" is defined as $u \cdot v = uv$.*

Definition 2.1.4. *Let $i \in \mathbb{N}_0$. An* i-th power of a word *w, denoted by w^i, is inductively defined in the following way:*

- $w^0 = \epsilon$,
- $w^i = w \cdot w^{i-1}$.

Next, we introduce the notion of a *language* and define some operations on languages.

Definition 2.1.5. *A* language *over alphabet Σ is a set of words over Σ. It is an arbitrary subset of Σ^*.*

Since each language is a set of words, we can perform standard set operations on them. Let L_1 and L_2 be two languages over alphabet Σ.

Definition 2.1.6. *The* union *of languages L_1 and L_2, denoted by $L_1 \cup L_2$, is defined as $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$.*

Definition 2.1.7. *The* intersection *of languages L_1 and L_2, denoted by $L_1 \cap L_2$, is defined as $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$.*

Definition 2.1.8. *The* complement *of language L_1, denoted by $\overline{L_1}$, is defined as $\overline{L_1} = \{w \mid w \in \Sigma^* \wedge w \notin L_1\}$.*

The following operations on words can be extended to languages:

Definition 2.1.9. *The* concatenation *of languages L_1 and L_2, denoted by $L_1 \cdot L_2$, is defined as $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$.*

Definition 2.1.10. *Let $i \in \mathbb{N}_0$. An* i-th power of a language *L, denoted by L^i is inductively defined as:*

- $L^0 = \{\epsilon\}$,
- $L^i = L \cdot L^{i-1}$.

With respect to the previous definition, we define the notions of *Kleene star* and *Kleene plus*.

Definition 2.1.11. *Kleene star/Kleene plus. Let L be a language. The* Kleene star *of L, denoted by L^*, is defined as*

$$L^* = \bigcup_{i=0}^{\infty} L^i.$$

The Kleene plus *of L, denoted by L^+, is defined as*

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

2.1.1 Example Languages

Throughout this thesis, we work with a number of interesting languages to demonstrate the difference in the expressive power of various register automata models, or to give an idea of some important properties of particular models. In the following, we give the list of these languages with their definitions, both formal and informal. We only use their names in the latter chapters.

- $L_{\exists repeat} = \{w \mid \exists i, j \colon i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\}$ denotes the language of words where *some symbol appears at least twice.*
- $L_{\neg\exists repeat} = \{w \mid \forall i, j \colon i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\}$ denotes the language of words where *each symbol appears at most once, i.e. no symbols repeat.*
- $L_{\exists,\neg\exists repeat} = L_{\exists repeat} \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot L_{\neg\exists repeat}$ denotes the language composed as the concatenation of $L_{\exists repeat}$ and $L_{\neg\exists repeat}$ with a delimiter.
- $L_{\forall repeat} = \{w \mid \forall i \exists j \colon i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\}$ denotes the language of words where *each symbol appears at least twice.*
- $L_{\neg\forall repeat} = \{w \mid \exists i \forall j \colon i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\}$ denotes the language of words where *there exists a symbol, that appears exactly once.*
- $L_{\exists a\text{-}no\text{-}b} = \{w \mid \exists i \colon a[w_i] \wedge \nexists j < i \colon b[w_j] \wedge \mathbb{D}[w_j] = \mathbb{D}[w_i]\}$ from [14, Proof of Proposition 3.2], denotes the language of words w such that *there exists an input element $\langle a, d\rangle$ that is not preceded by an occurrence of a $\langle b, d\rangle$ element.*
- $L_{\neg\exists a\text{-}no\text{-}b} = \{w \mid \forall i \colon a[w_i] \implies (\exists j < i \colon b[w_j] \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j])\}$ denotes the language of words where *each input element $\langle a, d\rangle$ is preceded by an occurrence of $\langle b, d\rangle$ element.*
- $L_{\forall a \exists b} = \big\{w \mid \forall i \colon a[w_i] \implies \big((\forall j \neq i \colon a[w_j] \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]) \wedge (\exists k > i \colon b[w_k] \wedge \mathbb{D}[w_i] = \mathbb{D}[w_k])\big)\big\}$ denotes the language of words where *no two a-positions contain the same data value and every a-position is followed by a matching b-position.*

2.2 Automata Theory

Finite automaton is a formal model that allows us to represent a class of infinite state systems in a finite way. Register automata, introduced as their generalization, were first presented under the term *finite memory automata* in the paper of Kaminski and Francez [22].

The difference between register automaton and finite automaton is that register automata are equipped with a finite number of registers that can store an arbitrary data value from the input tape, which allows for working with infinite alphabets. Finite automata only have bounded memory in the form of control states. If we were to work with infinite alphabets in the context of finite automata, the model could not tell infinitely many elements apart. On the other hand, because of their finiteness, finite automata enjoy many favorable properties, such as closure under Boolean operations, projections, homomorphism, etc. In the following sections, we introduce some of the formal models, each of which will be referred to in the latter chapters of this thesis. We begin by introducing some concepts crucial for understanding the motivation behind the phenomena that we deal with throughout this work.

Determinism and nondeterminism. A *deterministic* computation works in a way such that when the machine is in a given state and reads the next input symbol, there is at most one next state – the computation is *determined*. In a *non-deterministic* computation, there is more than one possibility of moving from the current state to the next state via the current symbol on the input tape. Intuitively, each step of computation follows a unique path from the preceding step, and for each word, there is only a single unique run that the automaton can make when reading its characters. Nondeterminism is a generalization of determinism, therefore each deterministic automaton is naturally nondeterministic. While nondeterminism is a useful concept in the theory of computation [41] leading to, among other things, compact representations of some languages, its use in practical applications may be inefficient. For example, when performing pattern matching based on some nondeterministic model, the need for trying out all possible runs over the given word may lead to an abundance of backtracking, thus leading to inefficient performance and malfunctions in the worst case scenario.

Infinite alphabet. The intuition behind infinite alphabets is not that the words accepted by the automaton may be of infinite length, but, rather, that the domain of characters of which the word consists is infinite.

Practical applications from which the needs for infinite alphabets arise include the need for concurrency, situations, when it is necessary to statically decide whether a particular program satisfies given specifications, or working with an unbounded number of processes, each of them having a unique identification, such that we have to maintain information about their interaction and cooperation. Other considerations leading to infinite alphabets arise from attempting to perform a classical model checking techniques on infinite state systems, or from the realm of databases [26].

Decidability of a problem. For a better understanding of the identified properties for the proposed model, we explain the notion of decidability. Certain problems can be solved algorithmically and others cannot. In latter chapters, our goal is to explore the limits of algorithmic solvability for some of the decision problems in case of the proposed model and its extensions. Intuitively, a *decision problem* P can be understood as a function,

whose range is the set {true, false}. That is, for any specified problem and a given input, the function returns *true* if the specified property is satisfied by the input, and it returns *false* otherwise. We say that a decision problem is *decidable*, if it can be solved by an algorithm that halts on all given inputs in a finite number of steps.

2.2.1 Finite Automaton

Finite automaton (FA) is a simple yet powerful formal model, with memory limited to its finite state space. Its function is to either *accept* or *reject* an input word, depending on whether it satisfies the pattern defined by a given FA.

First, we give the formal definition of a finite automaton, and in the following sections, we introduce formal models that are more powerful in their expressive power. However, they are built upon extending the structure of FA.

Definition 2.2.1. *A* finite automaton (FA) *is a tuple* $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$*, where:*

- Q *is a finite set of* states*,*
- Σ *is a finite non-empty* alphabet*,*
- Δ *is a* transition function*, such that* $\Delta\colon Q \times \Sigma \to 2^Q$*,*
- $I \subseteq Q$ *is a set of* initial states*, and*
- $F \subseteq Q$ *is a set of* final states*.*

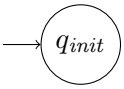
Definition 2.2.2. *A* configuration *of* $\mathcal{A}$ *is a pair* $c \in Q \times \Sigma^*$*, i.e., it consists of a state and the remaining substring of the processed word on the input tape. An* initial configuration *is a pair* $c \in I \times \Sigma^*$*. Suppose* $c_1 = (q_1, w)$*, and* $c_2 = (q_2, w')$ *are two configurations of* $\mathcal{A}$*. We say that* c_1 *can make a* step *to* c_2 *over* a*, denoted as* $c_1 \vdash^a c_2$*, iff*

1. $w = aw'$*,*
2. $q_2 \in \Delta(q_1, a)$*.*

A run ρ *of* $\mathcal{A}$ *over the word* $w = a_1a_2a_3 \dots a_n$ *from a configuration* c *is a sequence of configurations* $\rho = c_0c_1 \dots c_n$ *such that* $\forall 1 \leq i \leq n : c_{i-1} \vdash^{a_i} c_i$ *and* $c_0 = c$*. We say that* ρ *is* accepting *if* c *is an initial configuration,* $c_n = (s, \epsilon)$*, and* $s \in F$*.*

Definition 2.2.3. *The language accepted by* $\mathcal{A}$*, denoted as* $\mathcal{L}(\mathcal{A})$*, is defined as* $\mathcal{L}(\mathcal{A}) = \{w \mid \mathcal{A} \text{ has an accepting run over } w\}$*.*

Graphical representation. When depicting finite automata, the states are represented by circles, interconnected by arrows which represent the transition. For a transition to be enabled, above each transition, there is a symbol that needs to be present on the input tape when the computation's current state is the source state of the transition. The initial and final states are depicted in the following way:



(a) Depiction of an initial state.

(b) Depiction of a final state.

Example 2.2.1. *The finite automaton in Figure 2.2 accepts all words both starting and ending with letter* a*.*

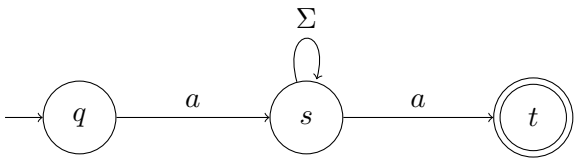


Figure 2.2: Example FA.

2.2.2 Register Automata

The model of register automaton lies at the basis of our introduced model. We build upon extending its structure, therefore, it is necessary to introduce a proper formal basis of RA. As mentioned before, the RA is a formal model operating over data words, whose data domain is an infinite set of values. It is equipped with a finite memory in the form of registers, each of which can store a single data value seen on the input tape.

Definition 2.2.4. *A (nondeterministic one-way)* register automaton *(on data words), abbreviated as (N)RA, is a tuple* $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ *where:*

- Q *is a finite set of* states*,*
- $\mathbf{R}$ *is a finite set of* registers*,*
- $I \subseteq Q$ *is a set of* initial states*,*
- $F \subseteq Q$ *is a set of* final states*, and*
- $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to \mathbf{R} \cup \{in, \bot\}) \times Q$ *is a* transition relation *such that if* $t\colon (q, a, g^{=}, g^{\neq}, up, s) \in \Delta$*, then* $g^{=} \cap g^{\neq} = \emptyset$*.*

We use $q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s$ to denote t (and often drop from up mappings $r \mapsto r$ for $r \in \mathbf{R}$, which we treat as implicit). The **semantics of** t is that:

- $\mathcal{A}$ can move from state q to state s if:
  - the Σ-symbol at the current position of the input word is a,
  - the $\mathbb{D}$-value at the current position is equal to all registers from $g^{=}$,
  - the $\mathbb{D}$-value at the current position is not equal to any register from $g^{\neq}$.
- The content of the registers is updated so that $r_i \leftarrow up(r_i)$ (i.e., r_i can be assigned the value of some other register, the current $\mathbb{D}$-symbol, denoted by in, or it can be cleared by being assigned $\bot$).

We refer to $g^{=}$ and $g^{\neq}$ as to *guard* of a transition, since it is a condition that needs to be satisfied in order to enable particular transition. In addition, *up* is referred to as *update*, since it denotes the way in which the content of registers gets updated after moving via given transition. In the following, we give the definitions for the formal basis of register automata.

Definition 2.2.5. *A* configuration *of* $\mathcal{A}$ *is a pair* $c \in Q \times (\mathbf{R} \to \mathbb{D} \cup \{\bot\})$*, i.e., it consists of a state and an assignment of data values to registers. An* initial configuration *of* $\mathcal{A}$ *is a pair* $c_{init} \in I \times \{\{r \mapsto \bot \mid r \in \mathbf{R}\}\}$*. Suppose* $c_1 = (q_1, f_1)$ *and* $c_2 = (q_2, f_2)$ *are two configurations of* $\mathcal{A}$*. We say that* c_1 *can make a* step *to* c_2 *over* $\langle a, d\rangle \in \Sigma \times \mathbb{D}$ *using transition* $t\colon q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s \in \Delta$*, denoted as* $c_1 \vdash_t^{\langle a,d\rangle} c_2$*, iff*

1. $d = f_1(r)$ *for all* $r \in g^{=}$,
2. $d \neq f_1(r)$ *for all* $r \in g^{\neq}$, *and*
3. *for all* $r \in \mathbf{R}$, *we have* $f_2(r) = \begin{cases} f_1(r') & \text{if } up(r) = r' \in \mathbf{R}, \\ d & \text{if } up(r) = in, \text{ and} \\ \bot & \text{if } up(r) = \bot. \end{cases}$

A run ρ *of* $\mathcal{A}$ *over the word* $w = \langle a_1, d_1 \rangle \dots \langle a_n, d_n \rangle$ *from a configuration* c *is a sequence of alternating configurations and transitions* $\rho = c_0 t_1 c_1 t_2 \dots t_n c_n$ *such that* $\forall 1 \leq i \leq n \colon c_{i-1} \vdash_{t_i}^{\langle a_i, d_i \rangle} c_i$ *and* $c_0 = c$.

We say that ρ is *accepting* if c is an initial configuration, $c_n = (s, f)$, and $s \in F$.

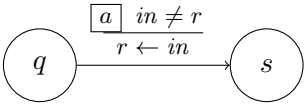
Definition 2.2.6. *The language accepted by* $\mathcal{A}$, *denoted as* $\mathcal{L}(\mathcal{A})$, *is defined as* $\mathcal{L}(\mathcal{A}) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathcal{A} \text{ has an accepting run over } w\}$.

A configuration c is *reachable* if there exists a run ρ starting from an initial configuration such that the last configuration of ρ is c. A state $q \in Q$ is *reachable* if there exists a reachable configuration (q, f) for some f. A configuration c is *backward-reachable* if there exists a run ρ starting from c such that the last configuration of ρ is (q, f) for some $q \in F$ and a state $q \in Q$ is *backward-reachable* if there exists a backward-reachable configuration (q, f) for some f. A state is *useful* if it is reachable and backward-reachable, it is *useless* otherwise.

We say that $\mathcal{A}$ is a *deterministic RA* (DRA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{=}, g_1^{\neq}, up_1} s_1, q \xrightarrow{a \mid g_2^{=}, g_2^{\neq}, up_2} s_2 \in \Delta$ we have that $g_1^{=} \cap g_2^{\neq} \neq \emptyset$ or $g_2^{=} \cap g_1^{\neq} \neq \emptyset$. $\mathcal{A}$ is *complete* if for all states $q \in Q$, symbols $a \in \Sigma$, and $g \subseteq \mathbf{R}$, there is a transition $q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s$ such that $g \subseteq g^{=}$ and $g \cap g^{\neq} = \emptyset$.

Properties. When it comes to decision problems of register automata, the non-emptiness of NRA is decidable [22]. The languages recognized by NRA are not closed under the complement, and, additionally, universality, together with equivalence and inclusion of NRA are undecidable [29]. One of the possibilities of regaining decidability is by restriction to deterministic register automata, which are closed under complement, and have a decidable inclusion, universality and equivalence [8]. The inclusion problem for $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, where $\mathcal{A}$ is an NRA and $\mathcal{B}$ is a DRA is decidable. The standard approach to testing $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ is to (i) determinise $\mathcal{B}$ into $\mathcal{B}^D$, (ii) complement $\mathcal{B}^D$ into $\overline{\mathcal{B}^D}$, (iii) test the emptiness of the intersection of $\mathcal{A}$ and $\overline{\mathcal{B}^D}$. Another way of gaining decidability is the restriction on the number of registers. The inclusion problem is decidable, when $\mathcal{A}$ is an NRA and $\mathcal{B}$ is an NRA with one register [22]. When increasing the number to two registers, the problem becomes undecidable again [10].

Graphical representation. When depicting register automata, the graphical representation of initial and final states is the same as for FA, with the exception in the depiction of the transition. Take, for example, the following transition:



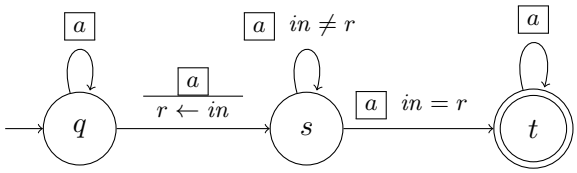
The RA can move from state q to state s iff:

1. the Σ-symbol at the current position is a, and

2. the $\mathbb{D}$-value (denoted by in in the picture) at the current position is different from the value stored in r.

The content of the register r is updated so that $r \leftarrow in$. This depiction is equivalent to the transition $q \xrightarrow{a \mid \emptyset, \{r\}, \{r \mapsto in\}} s$. That is, the symbol in the box denotes the current Σ-symbol, the formula above the line represents the guard of the transition, and the assignment below the line represents the update of the transition.

Example 2.2.2. *The* NRA *recognizing the language $L_{\exists repeat}$ can be seen in the following figure:*



2.2.3 Universal Register Automata

A *universal register automaton* (URA) is defined in the same way as a non-deterministic register automaton, with the exception of its accepting condition. In order for a word to be accepted by URA, each of its runs over this word has to be accepting.

Definition 2.2.7. *A language accepted by a* URA $\mathcal{A}_U$ *is the set* $\mathcal{L}(\mathcal{A}_U) = \{w \in (\Sigma \times \mathbb{D})^* \mid$ *every run of* $\mathcal{A}_U$ *on* w *is accepting*$\}$. *Note that the* URA *should be complete.*

There is a duality between NRAs and URAs.

Fact 2.2.1. *For every* NRA $\mathcal{A}_N$*, there is a* URA *accepting the complement of* $\mathcal{L}(\mathcal{A}_N)$. *Conversely, for every* URA $\mathcal{A}_U$*, there is an* NRA *accepting the complement of* $\mathcal{L}(\mathcal{A}_U)$.

Proof. For both parts, the complement automaton is obtained by

(i) adding a rejecting *sink* state to the automaton,

(ii) completing the transition relation (i.e., adding transitions for undefined combinations of symbols and guards to the sink state) of the input automaton, and

(iii) swapping final and non-final states.

□

Example 2.2.3. URA*s can, for instance, accept the language $L_{\neg\exists repeat}$ obtained as the complement of $L_{\exists repeat}$ from Example 2.2.2 (i.e., it is the language of words where no two data values are the same), which* NRA*s cannot accept.* URA*s on the other hand cannot accept $L_{\exists repeat}$. The* URA *accepting $L_{\neg\exists repeat}$ can be seen in Figure 2.3.*

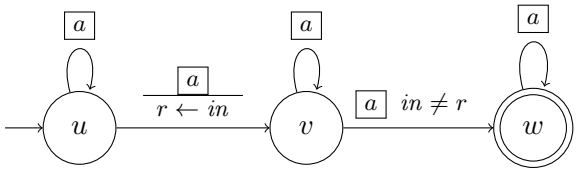


Figure 2.3: URA recognizing $L_{\neg\exists repeat}$.

2.2.4 Alternating Register Automata

An alternating register automaton makes use of *alternation*, which is a generalization of non-determinism. The difference between the two lies in the accepting conditions. The states of ARA are either *existential* (denoted by $\vee$), or *universal* (denoted by $\wedge$). When depicting ARAs, the transitions originating in a universal state are connected by an arc with the guard that needs to hold for all interconnected transitions. Existential transitions are denoted in the usual manner. An example of such an automaton can be seen in Figure 2.4. Note that in this picture, the state 1 is universal, even though it may look like it is both universal and existential. The two sets of transitions are distinguished by the different labels a and b required to enable the transitions.

Intuitively, in case of an existential state, for a word to be accepted from it, a condition *there exists at least one run originating in this state that accepts* has to hold. In order to accept a word from a universal state, the condition *all runs originating in this state have to accept* must hold. The run for ARAs is not linear (in a single thread), but branching into multiple threads, i.e., it is a tree. An example of the run of the ARA from Figure 2.4 over the data word $\langle b,4\rangle\langle a,1\rangle\langle b,2\rangle\langle a,3\rangle\langle b,1\rangle\langle b,3\rangle$ can be seen in Figure 2.5. Labels of the form $r = 1$ below the states denote the content of a register active for a given branch of computation.

It is obvious that alternating automata combine the accepting conditions of NRA and URA. A universal register automaton can be thought of as an ARA with only universal states, and a non-deterministic register automaton can be thought of as an ARA with only existential states.

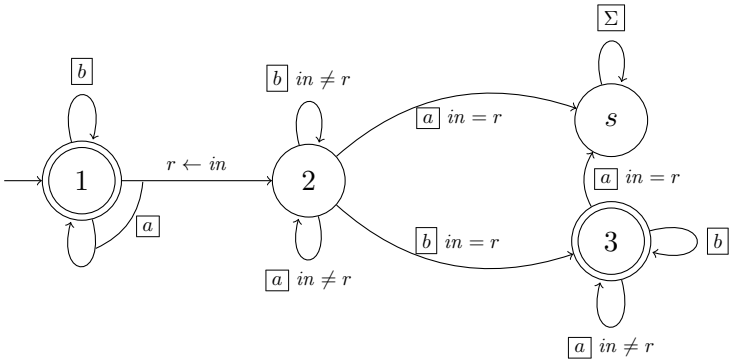


Figure 2.4: Example ARA, recognizing $L_{\forall a\exists b}$.

Definition 2.2.8. *An* alternating RA *(*ARA*) is a tuple* $\mathcal{A} = (Q = Q_\wedge \cup Q_\vee, \mathbf{R}, \Delta = \Delta_\wedge \cup \Delta_\vee \cup \Delta_\epsilon, I, F)$ *where* $Q_\wedge$ *and* $Q_\vee$ *denote the (disjoint) finite sets of* universal *and* existential *states respectively,* I *and* F *are the same as for* NRA*, and the transitions are as follows:*

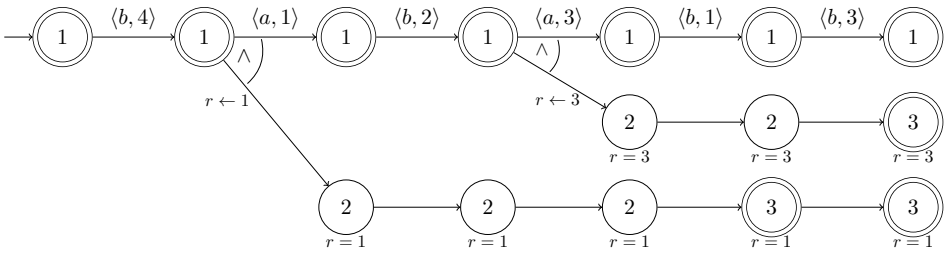


Figure 2.5: An example of a run of the ARA from Figure 2.4.

1. $\Delta_\wedge \subseteq Q_\wedge \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to \mathbf{R} \cup \{in, \bot\}) \times Q$,
2. $\Delta_\vee \subseteq Q_\vee \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to \mathbf{R} \cup \{in, \bot\}) \times Q$, *and*
3. $\Delta_\epsilon \subseteq Q \times Q$.

The transitions in Δ_ϵ are the so-called ϵ-transitions, which neither read any symbol from the input nor manipulate registers.

In this thesis, we only use the ARA to position our model in the landscape of expressivity of register automata models. Therefore, we refrain from giving other fully involved, technical definitions. These can be found e.g. in the paper of Demri and Lazić [10].

2.2.5 Alternating Register Automata with *guess* and *spread*

There are many possible ways of extending alternating register automata, in order to increase their expressivity or obtain other advantages in terms of their computational power. In this thesis, we also consider an extension introduced in [14].

In the aforementioned paper, authors explore the model of alternating register automata with *guess* and *spread* operations (ARA(guess, spread)). The *guess* operation allows a thread to assign a nondeterministically chosen data value into some register, which may be different for each thread. In addition, the *spread* operation allows to make a certain kind of universal quantification over the data values seen so far on the run of the automaton, including data values seen on different threads.

We explore the expressive power of ARA(guess, spread) in comparison with the expressivity of our proposed model. As with ARA, we do not directly work with this model. The overview of its formal basis, and the relation to its equivalent logic fragment can be found in [14].

By notation $\mathrm{NRA}_n, \mathrm{URA}_n$, and RsA_n, we denote the subclasses of respective register automata models with n registers. The $\mathrm{NRA}^=$, $\mathrm{URA}^=$, and $\mathrm{DRA}^=$ are used to denote the subclasses of NRAs, URAs, and DRAs with *no inequality* guards, i.e., automata where for every transition $q \xrightarrow{a \mid g^=, g^\neq, up} s$ it holds that $g^\neq = \emptyset$. Furthermore, for a class $\mathcal{C}$ of automata with registers and $n \in \mathbb{N}$, we use $\mathcal{C}_n$ to denote the subclass of $\mathcal{C}$ containing automata with at most n registers (e.g., DRA_2). We abuse notation and use $\mathcal{C}$ to also denote the class of languages defined by $\mathcal{C}$.

2.3 Regular Expressions

In theory of computation, formal languages can be described by expressions, which are built up by regular operations [41]. Such expressions are called *regular expressions*. Regular expression (regex) is a powerful mechanism for representation of regular languages, or a strong technique for describing patterns, first introduced by S. C. Kleene in [23]. Intuitively, each regex represents a formal structure of each word belonging to a language it represents.

Regexes are convenient in many practical applications, such as in data processing, natural language processing, pattern matching, data extraction, or even web scraping. In the following, we give their formal definition, followed by a definition of their possible extension with greater expressive power.

Definition 2.3.1. *Let Σ be an alphabet. A regular expression (regex) $\mathcal{R}$ over Σ is recursively defined in the following way:*

1. $\mathcal{R} = \emptyset$,
2. $\mathcal{R} = \epsilon$,
3. $\mathcal{R} = a$ *for some* $a \in \Sigma$,
4. $\mathcal{R} = \mathcal{R}_1 + \mathcal{R}_2$, *where* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *are regular expressions,*
5. $\mathcal{R} = \mathcal{R}_1\mathcal{R}_2$, *where* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *are regular expressions, and*
6. $\mathcal{R} = \mathcal{R}_1^*$, *where* $\mathcal{R}_1$ *is a regular expression.*

Regular expressions and finite automata are equivalent in their expressive power [18]. Any finite automaton can be converted into an equivalent regex and vice versa.

2.3.1 Regular Expressions with Backreferences

There are many important applications in which the usage of regular expressions is crucial. In some of them, the context calls for definition of more complicated patterns that cannot be represented by the general structure of regular expressions. One of them is manipulation with an arbitrary *substring* or a *subpattern* of some matched word. In this thesis, we focus on regular expressions with backreferences, since their processing is one of the main motivations for introduction of our proposed model. Intuitively, regular expressions with backreferences are used when one wishes to match the same piece of text more than once.

In most applications, the regular expression contains one or more *capture groups*, which enclose one or more characters by parentheses (e.g. `(.)`, which is a group capturing a single arbitrary character). Later in the regular expression, this capture group is referred to by a *backreference*, which is denoted by a backslash, followed by a particular number i (e.g. `\1`). Since there may be more than one capture groups, the number i denotes the i-th capture group from the beginning of the extended regular expression. The backreference requires the repetition of a particular pattern enclosed in the capture group appearing on the position of the backreference.

Example 2.3.1. *The extended regex* `/(.).*\1/` *captures words in which some symbol appears at least twice. It is equivalent to the language* $L_{\exists repeat}$ *from Example 2.2.2.*

Example 2.3.2. *The extended regex* `/(.).*(.).*\2.*\1/` *captures words in which arbitrary two symbols appear at least twice, such that their order is reversed in the second appearance.*

Chapter 3

Register Set Automaton

This chapter contains proper formal definitions of the introduced model and its properties. The proposed model, called *register set automaton*, is based on extending the model of register automaton, and allowing for storage of a set of values in each register, contrary to the singleton value in the original model.

In our work, we were inspired by *counting-set automata* introduced by Turoňová *et al.* [44]. These use *sets of counter values* to compactly represent configurations of *counting automata* [17] (a restricted version of counter automata [28] with a bound on the value of counters for compact representation of finite automata) in order to obtain a deterministic model for efficient matching of regular expressions with repetitions.

Definition 3.0.1. *A (nondeterministic)* register set automaton *(on data words), abbreviated as (N)*RsA *is a tuple* $\mathcal{A}_S = (Q, \mathbf{R}, \Delta, I, F)$ *where:*

- Q *is a finite set of* states*,*
- $\mathbf{R}$ *is a finite set of* registers*,*
- $I \subseteq Q$ *is a set of* initial states*,*
- $F \subseteq Q$ *is a set of* final states*, and*
- $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$ *is a set of* transitions *such that if* $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$*, then* $g^{\in} \cap g^{\notin} = \emptyset$ *(as with* NRA*s, we often do not write mappings* $r \mapsto \{r\}$ *for* $r \in \mathbf{R}$ *when defining the* update *(up)).*

The **semantics of a transition** $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s$ is the following:

- $\mathcal{A}_S$ can move from state q to state s if
  - the Σ-symbol at the current position of the input word is a,
  - the $\mathbb{D}$-value at the current position is in all registers from $g^{\in}$, and
  - the $\mathbb{D}$-value at the current position is in no registers from $g^{\notin}$.
- The content of the registers is updated so that $r_i \leftarrow \bigcup\{x \mid x \in up(r_i)\}$ (i.e., r_i can be assigned the union of values of several registers, possibly including the current $\mathbb{D}$-symbol denoted by in).

Definition 3.0.2. *A* configuration *of* $\mathcal{A}_S$ *is a pair* $c \in Q \times (\mathbf{R} \to 2^{\mathbb{D}})$*, i.e., it consists of a state and an assignment of sets of data values to registers.*

Definition 3.0.3. *An* initial configuration *of $\mathcal{A}_S$ is a pair $c_{init} \in I \times \{\{r \mapsto \emptyset \mid r \in \mathbf{R}\}\}$. Suppose $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ are two configurations of $\mathcal{A}_S$. We say that c_1 can make a* step *to c_2 over $\langle a, d\rangle \in \Sigma \times \mathbb{D}$ using transition $t\colon q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$, denoted as $c_1 \vdash_t^{\langle a,d\rangle} c_2$, iff*

1. *$d \in f_1(r)$ for all $r \in g^{\in}$,*
2. *$d \notin f_1(r)$ for all $r \in g^{\notin}$, and*
3. *for all $r \in \mathbf{R}$, we have $f_2(r) = \bigcup\{f_1(r') \mid r' \in \mathbf{R}, r' \in up(r)\} \cup \begin{cases} \{d\} & \text{if } in \in up(r) \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$*

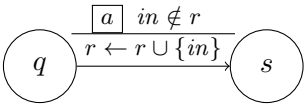
A run *ρ of $\mathcal{A}_S$ over the word $w = \langle a_1, d_1\rangle \dots \langle a_n, d_n\rangle$ from a configuration c is a sequence of alternating configurations and transitions $\rho = c_0 t_1 c_1 t_2 \dots t_n c_n$ such that $\forall 1 \leq i \leq n\colon c_{i-1} \vdash_{t_i}^{\langle a_i,d_i\rangle} c_i$ and $c_0 = c$.*

We say that ρ is *accepting* if c is an initial configuration, $c_n = (s, f)$, and $s \in F$.

Definition 3.0.4. *The language accepted by $\mathcal{A}_S$, denoted as $\mathcal{L}(\mathcal{A}_S)$, is defined as $\mathcal{L}(\mathcal{A}_S) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathcal{A}$ has an accepting run over $w\}$.*

We say that the RsA $\mathcal{A}_S$ is *deterministic* (DRsA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1, q \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2 \in \Delta$ we have that $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_2^{\in} \cap g_1^{\notin} \neq \emptyset$.

Graphical representation. When depicting register set automata, the graphical representation of initial and final states is, again, the same as for FA, with the exception in the depiction of the transition. Take, for example, the following transition:



The RsA can move from state q to state s iff:

1. the Σ-symbol at the current position is a,
2. the $\mathbb{D}$-value at the current position (denoted by *in* in the picture) does not belong to the register set r.

The content of the register r is updated so that $r \leftarrow r \cup \{in\}$. This depiction is equivalent to the transition $q \xrightarrow{a \mid \emptyset, \{r\}, \{r \mapsto \{r, in\}\}} s$. That is, the symbol in the box denotes the current Σ-symbol, the notion above the line represents the guard of the transition, and the notion below the line represents the update of the transition.

Example 3.0.1. *A* DRsA *accepting the language $L_{\exists repeat}$ from Example 2.2.2 can be seen in Figure 3.1. Formally, it is a* DRsA$_1$ *$\mathcal{A} = (\{q, s\}, \{r\}, \Delta, \{q\}, \{s\})$ where Δ contains the following transitions: $\Delta = \{q \xrightarrow{a \mid \emptyset, \{r\}, \{r \mapsto \{r, in\}\}} q,\ q \xrightarrow{a \mid \{r\}, \emptyset, \emptyset} s,\ s \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} s\}$. Intuitively, the* DRsA *waits in q and accumulates the so-far seen input data values in register r (we use $r \leftarrow r \cup \{in\}$ to denote the update $r \mapsto \{r, in\}$). Once the* DRsA *reads a value that is already in r, it moves to s and accepts.*

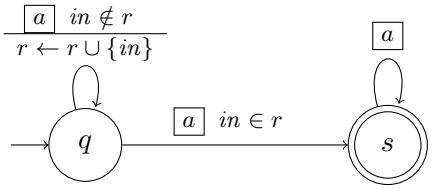


Figure 3.1: DRsA$_1$ recognizing $L_{\exists repeat}$.

Example 3.0.2. *Consider the language $L_{\neg\forall repeat}$. Intuitively, it is the language of all words containing a data value with exactly one occurrence. This language is accepted, e.g., by the* RsA$_1$ *in Figure 3.2. The* RsA *stays in state q, collecting the so-far seen values into its register, and at some point, when it encounters a value occurring for the first time, it nondeterministically moves to s, remembering the value in its register. Then, at state s, the* RsA *just checks that the previously stored value does not appear at the input any more.*

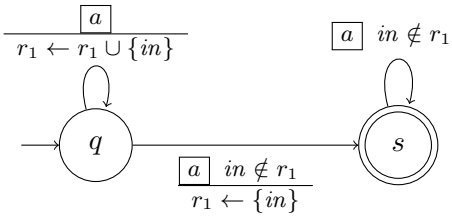


Figure 3.2: RsA$_1$ recognizing $L_{\neg\forall repeat}$.

Example 3.0.3. *Consider the extended regular expression $R_{\backslash 3 \backslash 2 \backslash 1}$ from Chapter 1. The* RsA *equivalent to this extended regex can be seen in Figure 3.3. Intuitively, it accumulates all encountered data values into the register r_1, until the `";"` delimiter is seen, then it accumulates all values in register r_2 while reaching another delimiter, and, finally, it accumulates all values into register r_3. At some point, the* RsA *non-deterministically checks whether the three last symbols belong to registers r_3, r_2 and r_1, respectively.*

Example 3.0.4. *A* DRsA$_1$ *accepting the language $L_{\neg\exists repeat}$, which is the complement of the language $L_{\exists repeat}$ from Example 2.2.2, can be seen in Figure 3.4.*

Intuitively, the automaton stays in state q and accumulates all input data values in register r, making sure no input data value has been seen previously.

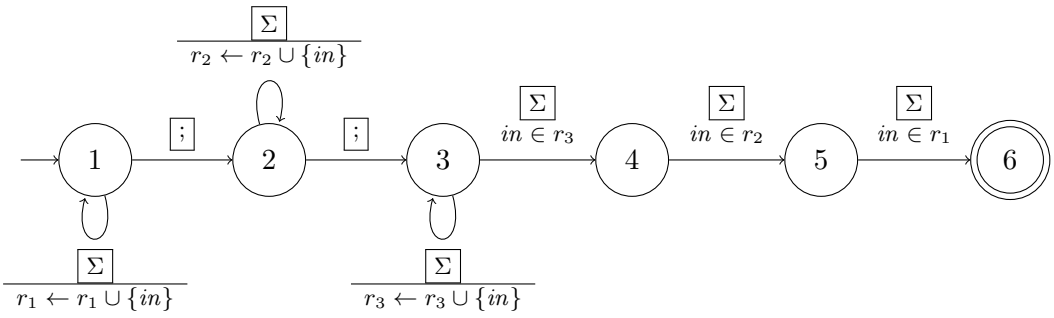


Figure 3.3: RsA equivalent to $R_{\backslash 3 \backslash 2 \backslash 1}$.

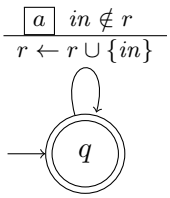


Figure 3.4: DRsA$_1$ recognizing $L_{\neg\exists repeat}$.

3.1 Properties

In this section, we discuss some of the properties for RsAs as well as the decidability of basic decision problems for RsAs. At first, we claim that RsAs generalize NRAs.

Fact 3.1.1. *For every $n \in \mathbb{N}$ and* NRA$_n$*, there exists an* RsA$_n$ *accepting the same language.*

Proof. We transform every NRA transition $q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s$ into the RsA transition $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up'} s$ such that $g^{\in} = g^{=}$, $g^{\notin} = g^{\neq}$, and for every register r_i and $up(r_i) = x$,

$$up'(r_i) = \begin{cases} \{x\} & \text{for } x \in \mathbf{R} \cup \{in\} \\ \emptyset & \text{for } x = \bot. \end{cases}$$

Intuitively, every simple register of NRA will be represented by a set register of RsA that will always hold the value of either an empty or a singleton set. □

The next theorem shows the core property of RsAs, which is that their emptiness problem is decidable. The proof of decidability and a proper classification of its complexity can be found in Chapter 4, which is entirely devoted to the emptiness checking of register set automata.

Theorem 3.1.1. *The emptiness problem for* RsA *is decidable, in particular,* $\mathbf{F}_\omega$*-complete.*

Remark 3.1.1. *Since register set automata generalize non-deterministic register automata, their universality, equivalence, and language inclusion problems are all undecidable.*

3.2 Closure Properties

In this section, we examine and prove the closure properties for RsA as well as for its more restricted versions, with respect to a fixed number of registers, and the property of determinism. We study the closure under Boolean operations, including union, intersection, and complement.

3.2.1 Closure Properties of RsA

The closure properties of RsAs are the same as for NRAs.

Theorem 3.2.1. *The following closure properties hold for* RsA*:*

1. RsA *is closed under union and intersection.*
2. RsA *is not closed under complement.*

Proof. The proofs for closure under union and intersection are standard: for two RsAs $\mathcal{A}_1 = (Q_1, \mathbf{R}_1, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \mathbf{R}_2, \Delta_2, I_2, F_2)$ with disjoint sets of states and registers, the RsA $\mathcal{A}_\cup$ accepting the union of their languages is obtained as $\mathcal{A}_\cup = (Q_1 \cup Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta_1 \cup \Delta_2, I_1 \cup I_2, F_1 \cup F_2)$. Similarly, $\mathcal{A}_\cap$ accepting their intersection is constructed as the product $\mathcal{A}_\cap = (Q_1 \times Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta', I_1 \times I_2, F_1 \times F_2)$ where

$$(s_1, s_2) \xrightarrow{a \mid g_1^{\in} \cup g_2^{\in}, g_1^{\notin} \cup g_2^{\notin}, up_1 \cup up_2} (s_1', s_2') \in \Delta'$$

iff

$$s_1 \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1' \in \Delta_1 \quad \text{and} \quad s_2 \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2' \in \Delta_2.$$

Correctness of the constructions is clear.

For showing the non-closure under complement, consider the language $L_{\neg\forall repeat}$ from Example 3.0.2, which can be accepted by RsA. Let us show that for the complement of the language, namely, the language $L_{\forall repeat}$, where all data values appear at least twice, there is no RsA that can accept it.

Our proof is a modification of the proof of Proposition 3.2 in [14]. In particular, we show that if $L_{\forall repeat}$ were expressible using an RsA, then we could construct an RsA encoding accepting runs of a Minsky machine. Since emptiness of an RsA is decidable (cf. Theorem 3.1.1) and emptiness of a Minsky machine is not, we would then obtain a contradiction.

Let us consider a Minsky machine $\mathcal{M}$ with two counters and instructions of the form (q, ℓ, q') where q and q' are states of $\mathcal{M}$ and $\ell \in \{\texttt{inc}, \texttt{dec}, \texttt{ifzero}\} \times \{1, 2\}$ is the corresponding counter operation. A run of $\mathcal{M}$ is a sequence of instructions (which can be viewed as Σ-symbols) together with a data value $d \in \mathbb{D}$ assigned to every symbol. The data values are used to match increments with decrements of the same counter (intuitively, we are trying to say that "*each increment is matched with a decrement*", in order to express that the value of the counter is zero). For instance, consider the following run:

$$\underset{12}{(q_1, \texttt{inc}_1, q_2)}\ \underset{42}{(q_2, \texttt{inc}_1, q_3)}\ \underset{12}{(q_3, \texttt{dec}_1, q_3)}\ \underset{17}{(q_3, \texttt{inc}_2, q_2)}\ \underset{42}{(q_2, \texttt{dec}_1, q_1)}\ \underset{7}{(q_1, \texttt{ifzero}_1, q_4)}$$

Here, the first increment of counter 1 is matched with the first decrement of the counter (both having data value 12) and the second increment of counter 1 is matched with the second decrement of the counter (both having data value 42). Since all increments of the counter are uniquely matched with a decrement, the test at the end is satisfied, so $\mathcal{M}$ would accept (we assume q_4 is a final state). To be able to accept such words, we can construct an automaton that checks the following properties of the input word:

1. The first instruction is of the form $(q_1, \cdot, \cdot)$ for q_1 being the initial state of $\mathcal{M}$.
2. Each instruction of the form $(\cdot, \cdot, q_i)$ is followed by an instruction of the form $(q_i, \cdot, \cdot)$.
3. All increments have different data values, and all decrements have different data values.
4. Between every two $(\cdot, \texttt{ifzero}_i, \cdot)$ instructions (or between the start and the first such an $\texttt{ifzero}_i$ instruction),
   (a) every $(\cdot, \texttt{dec}_i, \cdot)$ needs to be preceded by an $(\cdot, \texttt{inc}_i, \cdot)$ instruction with the same data value and

(b) every $(\cdot, \texttt{inc}_i, \cdot)$ needs to be followed by a $(\cdot, \texttt{dec}_i, \cdot)$ instruction with the same data value.

Properties 1 and 2 can be easily expressed using an NFA and, therefore, also using a DRsA. Property 3 is easily expressible using an RsA (in fact, using a DRsA) that collects data values of increments and decrements of each counter in registers (we need two registers for every counter). Property 4a is also expressible using an RsA (again, using a DRsA) that collects the data values of decrements and whenever it reads an increment, it checks whether it has seen the increment's data value before.

Let us now focus on Property 4b. The negation of this property would be "*there is an increment not followed by a decrement with the same data value*". This negated property is essentially captured by the language $L_{\neg\forall repeat}$ and so it is expressible using RsA (in fact, it can be expressed by an NRA with guessing; or by a simple NRA provided that we change the accepted language to be prepended by a sequence of data values that will be used in the run, separated from the run by a delimiter). Therefore, if an RsA could accept the complement of $L_{\neg\forall repeat}$, i.e., the language $L_{\forall repeat}$, then we would be able to solve the emptiness problem of a Minsky machine, which is a contradiction. □

3.2.2 Closure Properties of RsA_n

For RsAs with a limited number of registers, we lose the closure under intersection.

Theorem 3.2.2. *For each $n \in \mathbb{N}$, the following closure properties hold for RsA_n:*

1. *RsA_n is closed under union.*
2. *RsA_n is not closed under intersection and complement.*

Proof. The proof of closure under union is the same as in the proof of Theorem 3.2.1 with the exception that the result uses only registers $\mathbf{R}_1$ (we assume $|\mathbf{R}_1| = n$): all references to registers $r \in \mathbf{R}_2$ are changed to references to $f(r)$ where $f\colon \mathbf{R}_2 \to \mathbf{R}_1$ is an injection.

To show non-closure under intersection, consider the two languages

$$\mathcal{L}_n^A = \{w \mid \forall i < n\colon \mathbb{D}[w_i] = \mathbb{D}[w_{|w|-i+1}]\}$$

and

$$\mathcal{L}_n^B = \{w \mid \forall n \leq i < 2n\colon \mathbb{D}[w_i] = \mathbb{D}[w_{|w|-i+1}]\}.$$

Intuitively, $\mathcal{L}_n^A$ is the language of words where the first n data values in the word are repeated (in the reverse order) at the end of the word and $\mathcal{L}_n^B$ is the language of words where the $(n+1)$-th to $2n$-th data values are repeated (also in the reverse order) at the $2n$-th to $(n+1)$-th position from the end. Both languages can be expressed via NRA_n, and therefore also via RsA_n. Their intersection is the language $\mathcal{L}_n^{AB} = \{w \mid \forall i < 2n\colon \mathbb{D}[w_i] = \mathbb{D}[w_{|w|-i+1}]\}$, which is the same as $\mathcal{L}_{2n}^A$ and clearly needs $2n$ registers.

Non-closure under complement follows from Theorem 3.2.1 (its proof uses RsA_1). □

3.2.3 Closure Properties of DRsA

Theorem 3.2.3. DRsA *is closed under union, intersection, and complement.*

Proof. The proof of closure of DRsA under union is standard. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two complete DRsAs, such that $\mathcal{A}_1 = (Q_1, \mathbf{R}_1, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \mathbf{R}_2, \Delta_2, I_2, F_2)$, and their sets of states and registers are disjoint. The DRsA $\mathcal{A}_\cup$ accepting the union of their languages is obtained as $\mathcal{A}_\cup = (Q_1 \times Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta', I_1 \times I_2, F')$ where

$$(s_1, s_2) \xrightarrow{a \mid g_1^{\in} \cup g_2^{\in},\, g_1^{\notin} \cup g_2^{\notin},\, up_1 \cup up_2} (s_1', s_2') \in \Delta'$$

iff

$$s_1 \xrightarrow{a \mid g_1^{\in},\, g_1^{\notin},\, up_1} s_1' \in \Delta_1 \quad \text{and} \quad s_2 \xrightarrow{a \mid g_2^{\in},\, g_2^{\notin},\, up_2} s_2' \in \Delta_2,$$

and $F' = \{(q_1, q_2) \mid q_1 \in F_1 \vee q_2 \in F_2\}$.

The construction of the DRsA $\mathcal{A}_\cap = (Q_1 \times Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta', I_1 \times I_2, F'_\cap)$ accepting the intersection of $\mathcal{L}(\mathcal{A}_1)$ and $\mathcal{L}(\mathcal{A}_2)$ is similar to the construction of $\mathcal{A}_\cup$, with the exception of $F'_\cap$, which is obtained as $F'_\cap = F_1 \times F_2$.

The complement of DRsA is obtained in the standard way by completing it and swapping final and non-final states. Since the automaton is already deterministic, the correctness of the construction is obvious. □

3.2.4 Closure properties of DRsA$_n$

Theorem 3.2.4. *For each $n \in \mathbb{N}$, the following closure properties hold for* DRsA$_n$*:*

1. DRsA$_n$ *is closed under complement.*
2. DRsA$_n$ *is not closed under union and intersection.*

Proof. The closure under complement is trivial (complete the DRsA and swap final and non-final states; no new register is introduced).

To show that DRsA$_n$ is not closed under intersection, we use languages $\mathcal{L}_n^A$ and $\mathcal{L}_n^B$ from the proof of Theorem 3.2.2. In particular, both these languages are in DRA$_n$ (the DRA$_n$ needs more states than the corresponding NRA$_n$ because it cannot *guess* where the final part of the word starts and needs to consider all posibilites, making the DRA$_n$ exponentially larger). Similarly as in the proof of Theorem 3.2.2, a DRsA for the intersection of the languages, the language $\mathcal{L}_n^{AB}$, would need at least $2n$ registers.

Non-closure under union follows from De Morgan's laws. □

3.3 The Power of Nondeterminism

As with RAs, nondeterminism also allows bigger expressivity for RsAs.

Theorem 3.3.1. DRsA $\subsetneq$ RsA

Proof. Let us consider the language $L_{\neg\forall repeat}$ from the proof of Theorem 3.2.1, which is expressible using RsAs, and its complement $L_{\forall repeat}$, which is not expressible using RsAs. Since DRsAs are closed under complement (Theorem 3.2.3), if they could accept $L_{\neg\forall repeat}$, they could also accept $L_{\forall repeat}$, which is a contradiction. Therefore, $L_{\neg\forall repeat} \notin$ DRsA. □

Chapter 4

Emptiness Problem

In this chapter, we examine another property of register set automata. Namely, we deal with the decidability of the *emptiness problem*. For a given automaton $\mathcal{A}$, this problem asks whether $\mathcal{A}$'s language is empty.

In the case of finite automata, this question can be easily answered by removing any unreachable states from the automaton and checking whether there are any final states left. On the other hand, in case of a register set automaton, this problem is much harder, since there are sequences of conditions on transitions that need to be true in order to reach the final state, in addition to update functions, which move data values between register sets after each step of a run. However, in the following sections we prove that this problem is indeed decidable, and we determine its particular complexity.

4.1 Complexity Classes

We begin with a short overview of how decision problems are classified. In order to understand how hard the particular problem of emptiness checking for RsAs is, we also give a brief introduction into the fast-growing hierarchy of functions.

Complexity classes are used as a standardized tool for classification and comparison of computational problems. Each class contains a set of problems which take similar range of time and space to solve [21]. Most of the classes involving problems that range from tractable[1] to intractable[2] can be observed in the well-guided *Complexity Zoo* [1]. One can classify the upper or lower complexity bound of a given problem by reducing to or from another problem whose complexity is already identified. List of well-suited problems for the reductions can be found in the botanical companion to the aforementioned Complexity Zoo, the so-called *Complexity Garden* [19].

At first, we give the definitions for notions related to defining a particular complexity of a computational problem P. Let C be a complexity class. Intuitively, a *reduction* is a computable function that converts instances of problem P to a computational problem S. Any instance of the problem P can then be solved by using the reduction to first convert it to an instance of the problem S, and then apply the solver for the problem S [41]. The notation $P \leq S$ is used to denote the reduction from decision problem P to problem S. Additionally, in theory of formal languages, words over suitable alphabet Σ are used to

[1] Problems solved by computer algorithms running in polynomial time.

[2] There do not exist any algorithms for solving of these problems in polynomial time.

represent individual instances of some decision problem P. Naturally, this means that a decision problem P can be represented by a formal language $\mathcal{L}_P$.

Definition 4.1.1. *Let $\mathcal{R}$ denote a class of functions. Language $\mathcal{L}_P$ is $\mathcal{R}$* reducible *to language $\mathcal{L}_S$ (also known as $\mathcal{R}$ many-to-one reducible), if there exists a function f from $\mathcal{R}$ such that*

$$w \in \mathcal{L}_P \iff f(w) \in \mathcal{L}_S.$$

Definition 4.1.2. *Problem P is said to be C-*hard *if every problem in the class C reduces to problem P. When it is possible to reduce problem P to problem S, problem P is at least as hard as problem S.*

Definition 4.1.3. *Problem P is said to be C-*complete*, if every problem in the class C reduces to problem P, and P is also in the class C itself.*

4.1.1 Beyond elementary

The work of Schmitz [35] was created with the goal of introducing complexity classes beyond elementary, ones that bring some structure between the classes of `Elem` and `PR`, as well as between the classes of `PR` and `R`. See [3] for introduction to `Elem`, `PR`, and `R`. Classes beyond elementary were introduced as a tool for proper classification of truly intractable problems, which can be found in the areas of logic, formal languages, or verification.

The aforementioned work introduced an ordinal-indexed hierarchy $(\mathbf{F}_\alpha)_\alpha$ of *fast-growing* complexity classes of nonelementary complexities. These complexity classes are related to extended Grzegorczyk [25] $(\mathcal{F}_\alpha)_\alpha$ hierarchies, which are well suited for characterization of various complexity classes including functions computed by forms of `for` programs or terminating `while` programs [27, 13]. The overview of some complexity classes together with ones introduced in the aforementioned work can be seen in Figure 4.1.

$\mathbf{F}_\omega$ class. Intuitively, this class corresponds to non-primitive recursive *Ackermannian* problems closed under primitive-recursive reductions.

Let `FPR` denote the set of primitive-recursive functions. In the following definition, we use the notion of `DTime`. Intuitively, `DTime` represents a particular computational resource of computation time. It is the class of decision problems solvable by a deterministic Turing machine [41] in time $\mathcal{O}(f(n))$ [1], where $\mathcal{O}$ is the asymptotic notation for estimating the running time of a given algorithm.

Formally, $\mathbf{F}_\omega$ is a class containing problems that are decidable with $\mathbf{F}_\omega$ resources of some primitive-recursive function of the input size

$$\mathbf{F}_\omega \stackrel{\text{def}}{=} \bigcup_{p \in \mathtt{FPR}} \mathtt{DTime}\Big(\mathbf{F}_\omega\big(p(n)\big)\Big).$$

Problems such as finite containment problem for vector addition systems [20], universality of one-dimensional vector addition systems [16], or the emptiness of ARA_1 [10] are complete for this class. See [40, 45, 39] for deeper explanation.

4.2 Transfer Petri Nets

Intuitively, transfer Petri net is an extension of Petri nets where transitions can *transfer* all tokens from one place to another place at once. They are closely related to *broadcast pro-*

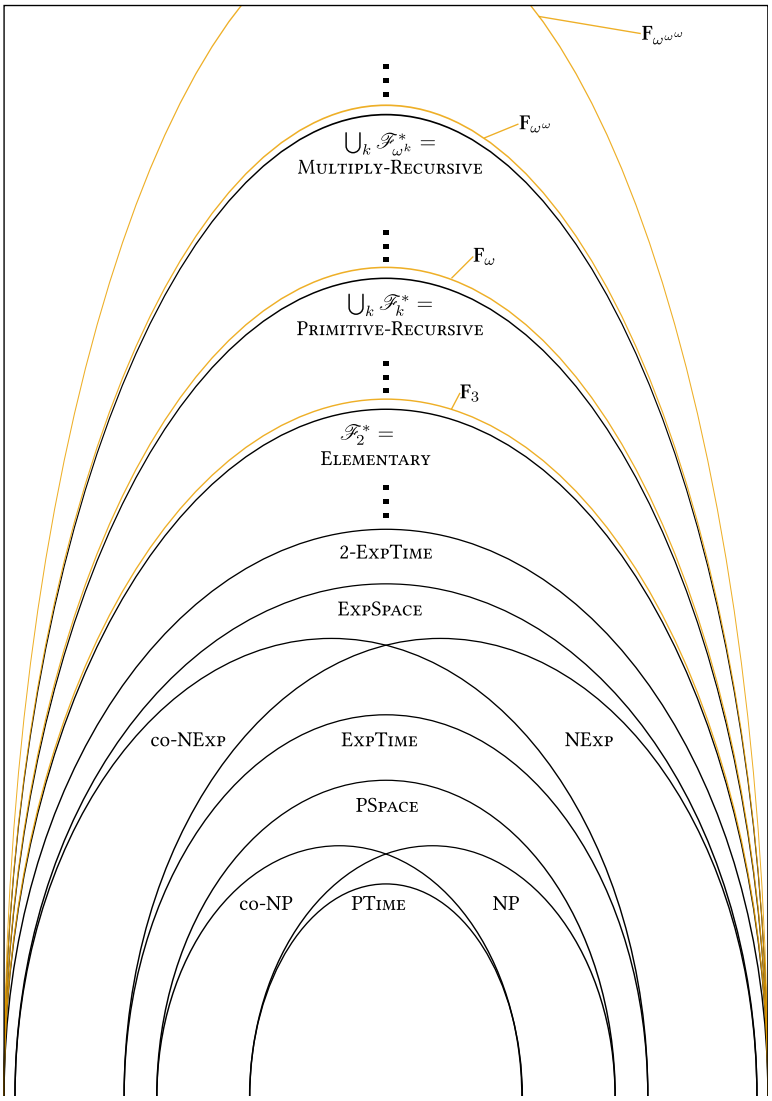


Figure 4.1: Some complexity classes. Adapted from [37].

tocols [12]. Broadcast protocols are appropriate for many applications, including analyzing bus-based hardware protocols such as ones designed for cache coherency [11].

Definition 4.2.1. *A* transfer Petri net *(TPN) is a triple* $\mathcal{N} = (P, T, M_0)$*, s.t.*

- P *is a finite set of* places*,*
- T *is a finite set of* transitions*, and*
- $M_0 \colon P \to \mathbb{N}$ *is an* initial marking*.*

The set of transitions T is such that $P \cap T = \emptyset$ and every transition $t \in T$ is of the form $t = \langle In, Out, Transfer \rangle$ where $In, Out \colon P \to \mathbb{N}$ define t's *input* and *output places* respectively and $Transfer \colon P \to P$ is a (total) *transfer function.*

Definition 4.2.2. *A* marking *of* $\mathcal{N}$ *is a function* $M \colon P \to \mathbb{N}$ *assigning a particular number of tokens to each place. Given a pair of markings* M *and* M'*, we use* $M \leq M'$ *to denote*

that for all $p \in P$ it holds that $M(p) \leq M'(p)$. Moreover, we use $\mathbf{u}_p$ to denote the marking such that $\mathbf{u}_p(p') = 1$ if $p = p'$ and $\mathbf{u}_p(p') = 0$ otherwise.

Given a marking M, we say that a transition $t = \langle In, Out, Transfer \rangle$ is *enabled* if $In \leq M$, i.e., there is a sufficient number of tokens in each of its input places. We use $M[t\rangle M'$ to denote that:

1. t is enabled in M and
2. M' is the marking such that for every $p \in P$ the following holds:

$$M'(p) = \sum\{M_{aux}(p') \mid Transfer(p') = p\} + Out(p), \text{ where } M_{aux} = M - In.$$

That is, the successor marking M' is obtained by

(i) removing In tokens from inputs of t,

(ii) transferring tokens according to $Transfer$, and

(iii) adding Out tokens to t's outputs.

We say that a marking M is *reachable* if there is a (possibly empty) sequence $t_1, t_2, \ldots, t_n$ of transitions such that it holds that $M_0[t_1\rangle M_1[t_2\rangle \ldots [t_n\rangle M$, where M_0 is the initial marking.

A marking M is *coverable* if there exists a reachable marking M', such that $M \leq M'$. The *Coverability* problem for TPNs asks, given a TPN $\mathcal{N}$ and a marking M, whether M is coverable in $\mathcal{N}$.

Proposition 4.2.1 ([38])**.** *The Coverability problem for TPN is $\mathbf{F}_\omega$-complete.*

4.3 Proof of $\mathbf{F}_\omega$-completeness of RsA emptiness

The next theorem shows that the emptiness problem of RsA is decidable, but for a much higher price than for NRAs, for which it is PSPACE-complete[3] [10]. For classifying the complexity of the problem, we use the hierarchy of fast-growing complexity classes of Schmitz [35] (cf. Section 4.1.1).

Theorem 4.3.1. *The emptiness problem for* RsA *is decidable. In particular, the problem is $\mathbf{F}_\omega$-complete.*

Intuition behind the proof. The proof is done by showing interreducibility of RsA emptiness with coverability in *transfer Petri nets* (TPNs) (often used for modelling the so-called *broadcast protocols*), which is a known $\mathbf{F}_\omega$-complete problem [38, 36, 37]. In the following, we briefly describe both directions of the reduction, followed by its formal definitions, proofs, and examples.

(RsA emptiness $\leq$ TPN coverability) The formal definition of the reduction can be seen in Section 4.3.1. Intuitively, the conversion of an RsA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ into a TPN $\mathcal{N}_\mathcal{A}$ is done in the following way. The set of places of $\mathcal{N}_\mathcal{A}$ will be as follows:

[3] Note that for an alternative definition of NRAs considered in [22, 33], where no two registers can contain the same data value, the problem is NP-complete [33].

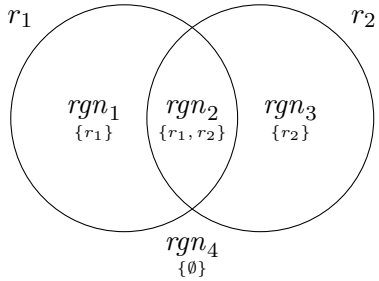


Figure 4.2: Venn diagram depicting possible regions for two registers of RsA.

(i) one place for each state of $\mathcal{A}$,

(ii) two special places *init* and *fin*, and

(iii) one place for every subset $rgn \subseteq \mathbf{R}$; these places are used to represent all possible intersections of values held in the registers. We call these intersections regions. For instance, if there are four tokens in the place representing $r_1 \cap r_2$, it means that there are exactly four different data values stored in both r_1 and r_2 and in no other register. Depiction of these regions is shown in Figure 4.2.

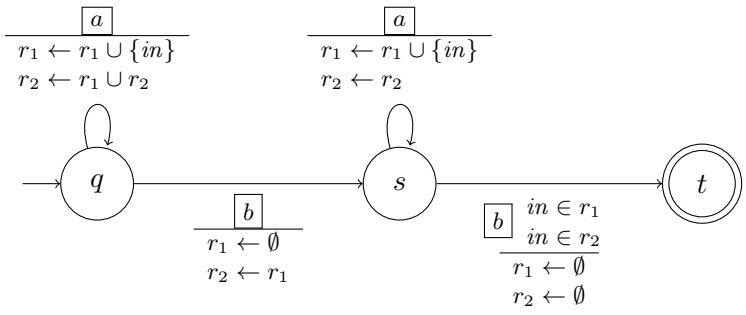


Figure 4.3: An example RsA.

Each transition t of $\mathcal{A}$ is simulated by one or more TPN transitions between places representing its source and target states. The number of respective TPN transitions depends on how specific the guard is in the original automaton, since we need to distinguish every possible option of *in* being in some region $rgn \in 2^{\mathbf{R}}$. The transitions move the token between the places corresponding to t's source and target states and, moreover, use the *broadcast* arcs to move tokens between the places representing regions, according to the manipulation of the set-registers in the update function of t. The special place *init* is used to have a single starting marking (it just nondeterministically chooses one state from I) and the place *fin* is used as the target for the coverability test; all places corresponding to final states of $\mathcal{A}$ can simply transition into it. An example of a reduced TPN equivalent to transition of the RsA in Figure 4.3 can be seen in Figure 4.4.

(TPN coverability $\leq$ RsA emptiness) The formal proof with definitions is in Section 4.3.2.

Given a TPN $\mathcal{N}$, the RsA $\mathcal{A}_\mathcal{N}$ simulating it will have the following structure. There will be a state q_{main}, which will be active before and after the simulation of firing

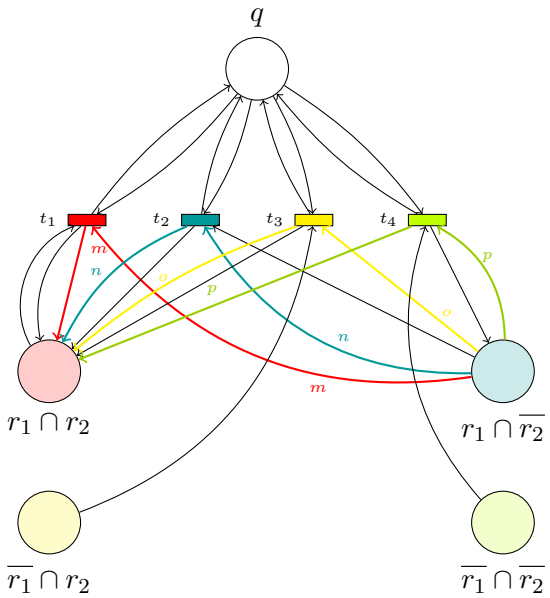


Figure 4.4: A TPN equivalent to the transition $q \xrightarrow{a \mid \emptyset, \emptyset, \{r_1 \mapsto \{r_1, in\}, r_2 \mapsto \{r_1, r_2\}\}} q$ from the RsA in Figure 4.3. Corresponding colors represent the position of the *in* value for a given transition.

each transition of $\mathcal{N}$. Moreover, there will be one register for every place of $\mathcal{N}$; individual tokens in the places will be simulated by unique data values from $\mathbb{D}$ stored in the corresponding registers. For each transition of $\mathcal{N}$, there will be a *gadget*, doing a cycle on q_{main}, that represents the semantics of $\mathcal{N}$'s transition. Each such gadget is composed of several *protogadgets*, which simulate basic actions performed during the transition (adding a token to a place, removing a token, moving all tokens between places). Implementation of adding a token and moving tokens is relatively easy, the tricky part is removing a token, since RsAs do not support removing a data value from a register. We solve this by using a *lossy remove*: i.e., if *one* token is to be removed from a place, we simulate it by removing *at least one* token (but potentially more). This will not preserve *reachability*, but it is enough to preserve *coverability*. Moreover, there will also be an *initial* part setting the contents of the registers to reflect the initial marking of $\mathcal{N}$ (terminating in q_{main}) and a *final* part that checks the coverability by removing (again in a lossy way) tokens from places, terminating in a single final state. □

In the following sections, we prove the two directions of the proof of Theorem 4.3.1.

4.3.1 Reduction from RsA emptiness to TPN coverability

In this section, we give the formal definition of the reduction from emptiness testing in register set automata to coverability in transfer Petri nets. The reduction is followed by a proof that shows that it preserves the answer of the reduced problem.

Lemma 4.3.1. *The emptiness problem for* RsA *is in* $\mathbf{F}_\omega$.

Proof. The proof of the lemma is based on reducing the RsA emptiness problem to coverability in TPNs, which is $\mathbf{F}_\omega$-complete (Proposition 4.2.1). Intuitively, the reduction consists

of creating a TPN with places representing both individual states of RsA and individual *regions* of the Venn diagram of $\mathbf{R}$. Transitions of RsA are represented by one or more transitions of TPN, distinguishing every possible option of *in* being in some region $rgn \in 2^{\mathbf{R}}$. The set of arcs leading to and from each transition is calculated in a way that preserves the semantics and position of values defined by the *guard* and *update* formulae. Finally, the marking to be covered requires one token to be present in the places representing the final states of the RsA.

Construction of $\mathcal{N}_\mathcal{A}$. Formally, let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be an RsA. In the following, we will construct a TPN $\mathcal{N}_\mathcal{A} = (P, T, M_0)$ and a marking M_F such that $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff M_F is coverable in $\mathcal{N}_\mathcal{A}$. We set the components of $\mathcal{N}_\mathcal{A}$ as follows:

- $P = Q \uplus \{init, fin\} \uplus 2^{\mathbf{R}}$ where *init* and *fin* are two new places,
- $T = \{\langle \mathbf{u}_{init}, \mathbf{u}_{q_i}, \mathbf{id}\rangle \mid q_i \in I\} \cup \{\langle \mathbf{u}_{q_f}, \mathbf{u}_{fin}, \mathbf{id}\rangle \mid q_f \in F\} \cup T'$ with T' defined below, and
- $M_0 = \mathbf{u}_{init}$.

Intuitively, the set of places contains the states of $\mathcal{A}$ (there will always be at most one token in those places), two new places *init* and *fin*, which are used for the initial nondeterministic choice of some initial state of $\mathcal{A}$ and for a unique *final place* (whose coverability will be checked) respectively, and, finally, a new place for every *region* of the Venn diagram of $\mathbf{R}$, which will track the number of data values that two or more registers share (e.g., for $\mathbf{R} = \{r_1, r_2, r_3\}$, the subset $\{r_1, r_3\}$ denotes the region $r_1 \cap \overline{r_2} \cap r_3$, i.e., the data values that are stored in r_1 and r_3 but are not stored in r_2. The region $\overline{r_1} \cap \overline{r_2} \cap \overline{r_3}$ is denoted as $\{\emptyset\}$. Regions rgn_1, rgn_2 are *distinct*, if $rgn_1 \triangle rgn_2 \neq \emptyset$, i.e. $\exists r\colon (r \in rgn_1 \cup rgn_2) \wedge r \notin rgn_1 \cap rgn_2$.

We now proceed to the definition of T'. Let $t = q \text{–}\boxed{a \mid g^{\in}, g^{\notin}, up}\!\!\rightarrow s \in \Delta$ be a transition in $\mathcal{A}$. Then, we create a TPN transition for every possible option of *in* being in some region $rgn_g \in 2^{\mathbf{R}}$ (e.g., for $in \in r_1 \cap \overline{r_2} \cap r_3$ or $in \in \overline{r_1} \cap \overline{r_2} \cap r_3$). For t and rgn_g, we define

$$\gamma(t, rgn_g) = \begin{cases} \{\langle In, Out, Transfer\rangle\} & \text{if } (g^{\in} \subseteq rgn_g) \wedge (g^{\notin} \cap rgn_g = \emptyset) \text{ and} \\ \emptyset & \text{otherwise.} \end{cases} \tag{4.1}$$

Then $T' = \bigcup\{\gamma(t, rgn_g) \mid t \in \Delta, rgn_g \in 2^{\mathbf{R}}\}$.

- $In = \mathbf{u}_{rgn_g} + \mathbf{u}_q$ and
- $Out = \mathbf{u}_{dst} + \mathbf{u}_s$ where $dst = \{r_i \in \mathbf{R} \mid in \in up(r_i)\}$.
- Before we give a formal definition of *Transfer*, let us start with an intuition given in the following example.

  Example 4.3.1. *Let us consider the register set automaton in Figure 4.3 and its transition $q \text{–}\boxed{a \mid \emptyset, \emptyset, up}\!\!\rightarrow q$ with $up(r_1) = \{r_1, in\}$ and $up(r_2) = \{r_1, r_2\}$. We need to update the following four regions of the Venn diagram of r_1 and r_2: $r_1 \cap r_2$, $r_1 \cap \overline{r_2}$, $\overline{r_1} \cap r_2$, and $\overline{r_1} \cap \overline{r_2}$. From the update function up, we see that the new values stored in r_1 and r_2 will be (we used primed versions of register names to denote their value after update) $r_1' = r_1$ (we do not consider $\{in\}$ here because it has been discharged within Out in the previous step) and $r_2' = r_1 \cup r_2$. The values of the regions will therefore be updated as follows:*

$$
\begin{aligned}
r_1' \cap r_2' &= r_1 \cap (r_1 \cup r_2) \\
&= (r_1 \cap r_1) \cup (r_1 \cap r_2) \\
&= r_1 \cup (r_1 \cap r_2) \\
&= (r_1 \cap \overline{r_2}) \cup (r_1 \cap r_2)
\end{aligned}
\qquad
\begin{aligned}
r_1' \cap \overline{r_2'} &= r_1 \cap \overline{(r_1 \cup r_2)} \\
&= r_1 \cap \overline{r_1} \cap \overline{r_2} \\
&= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\overline{r_1'} \cap r_2' &= \overline{r_1} \cap (r_1 \cup r_2) \\
&= (\overline{r_1} \cap r_1) \cup (\overline{r_1} \cap r_2) \\
&= \overline{r_1} \cap r_2
\end{aligned}
\qquad
\begin{aligned}
\overline{r_1'} \cap \overline{r_2'} &= \overline{r_1} \cap \overline{(r_1 \cup r_2)} \\
&= \overline{r_1} \cap \overline{r_1} \cap \overline{r_2} \\
&= \overline{r_1} \cap \overline{r_2}
\end{aligned}
\tag{4.2}
$$

Note that in the last step of the calculation of $r_1' \cap r_2'$, we used the fact that $r_1 = (r_1 \cap r_2) \cup (r_1 \cap \overline{r_2})$ in order to obtain a union of regions. From the previous calculation, we see that Transfer should be set as follows: $Transfer(\{r_1, r_2\}) = \{r_1, r_2\}$, $Transfer(\{r_1\}) = \{r_1, r_2\}$, $Transfer(\{r_2\}) = \{r_2\}$, and $Transfer(\{\emptyset\}) = \{\emptyset\}$. ◁

Computation of *Transfer* function. Formally, *Transfer* is computed as follows. For every $rgn_o \in 2^{\mathbf{R}}$, let us compute the sets of sets of registers

$$pst_\Pi(rgn_o) = \{up(r_i) \cap \mathbf{R} \mid r_i \in rgn_o\} \quad \text{and} \quad ngt(rgn_o) = \bigcup \{up(r_i) \cap \mathbf{R} \mid r_i \notin rgn_o\} \tag{4.3}$$

(*pst* is for "*positive*" and *ngt* is for "*negative*", which represent registers that occur positively and negatively, respectively, in the specification of the region of the Venn diagram rgn_o). The intuition is that $pst_\Pi(rgn_o)$ represents the update of rgn_o as the *product of sums* (intersection of unions), cf. the first line of Equation 4.2 in Example 4.3.1. Next, we convert the product of sums $pst_\Pi(rgn_o)$ into a sum of products

$$pst_\Sigma(rgn_o) = \coprod pst_\Pi(rgn_o) \tag{4.4}$$

where $\coprod\{D_1, \ldots, D_n\}$ is the *unordered Cartesian product* of sets $D_1, \ldots, D_n$, i.e.,

$$\coprod\{D_1, \ldots, D_n\} = \big\{\{d_1, \ldots, d_n\} \;\big|\; (d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n\big\}. \tag{4.5}$$

Example 4.3.2. *In the transition considered in Example 4.3.1, we obtain the following:*

$$
\begin{aligned}
pst_\Pi(\{r_1, r_2\}) &= \{\{r_1\}, \{r_1, r_2\}\} \\
pst_\Sigma(\{r_1, r_2\}) &= \{\{r_1\}, \{r_1, r_2\}\} \\
ngt(\{r_1, r_2\}) &= \emptyset
\end{aligned}
\qquad
\begin{aligned}
pst_\Pi(\{r_1\}) &= \{\{r_1\}\} \\
pst_\Sigma(\{r_1\}) &= \{\{r_1\}\} \\
ngt(\{r_1\}) &= \{r_1, r_2\}
\end{aligned}
$$

$$
\begin{aligned}
pst_\Pi(\{r_2\}) &= \{\{r_1, r_2\}\} \\
pst_\Sigma(\{r_2\}) &= \{\{r_1\}, \{r_2\}\} \\
ngt(\{r_2\}) &= \{r_1\}
\end{aligned}
\qquad
\begin{aligned}
pst_\Pi(\{\emptyset\}) &= \{\{\emptyset\}\} \\
pst_\Sigma(\{\emptyset\}) &= \{\{\emptyset\}\} \\
ngt(\{\emptyset\}) &= \{r_1, r_2\}
\end{aligned}
$$

◁

Next, we modify pst_Σ into pst_Σ' by removing regions that are incompatible with *ngt* to obtain

$$pst_\Sigma'(rgn_o) = \{x \in pst_\Sigma(rgn_o) \mid x \cap ngt(rgn_o) = \emptyset\}. \tag{4.6}$$

Example 4.3.3. *In the running example, we would obtain the following values of pst_Σ':*

$$
\begin{aligned}
pst_\Sigma'(\{r_1, r_2\}) &= \{\{r_1\}, \{r_1, r_2\}\} \\
pst_\Sigma'(\{r_2\}) &= \{\{r_2\}\}
\end{aligned}
\qquad
\begin{aligned}
pst_\Sigma'(\{r_1\}) &= \emptyset \\
pst_\Sigma'(\{\emptyset\}) &= \{\{\emptyset\}\}
\end{aligned}
$$

Compare the results with the calculation in Equation 4.2. ◁

Lastly, for every $rgn_i \in 2^{\mathbf{R}}$ such that $pst'_\Sigma(rgn_o) \in rgn_i$, we set $Transfer(rgn_i) = rgn_o$.

Example 4.3.4. *Continuing in the running example, we obtain*

$$Transfer(\{r_1, r_2\}) = \{r_1, r_2\} \qquad Transfer(\{r_1\}) = \{r_1, r_2\}$$
$$Transfer(\{r_2\}) = \{r_2\} \qquad Transfer(\{\emptyset\}) = \{\emptyset\},$$

which is the same result as in Example 4.3.1. Figure 4.4 contains the TPN fragment for all TPN transitions constructed from $\mathcal{A}$'s transition $q \xrightarrow{a \mid \top, \top, \{r_1 \mapsto \{r_1, in\}, r_2 \mapsto \{r_1, r_2\}\}} q$. ◁

The following claim shows that this construction is indeed well defined.

Claim 4.3.1. *The function Transfer is well defined.*

Proof. It is necessary to show that no set of values will be duplicated and assigned to two distinct regions when *Transfer* is calculated. According to the definition of *Transfer*, for all regions $rgn' \in 2^{\mathbf{R}}$ such that $pst'_\Sigma(rgn_o) \in rgn'$, the value of $Transfer(rgn')$ is set to be rgn_o, therefore we need to prove that for each pair of distinct regions rgn_1 and $rgn_2 \in 2^{\mathbf{R}}$ it holds that $pst'_\Sigma(rgn_1) \cap pst'_\Sigma(rgn_2) = \emptyset$. We prove this by contradiction.

Assume that there are two distinct regions rgn_1 and rgn_2 such that there exists a region $rgn_3 \in pst'_\Sigma(rgn_1) \cap pst'_\Sigma(rgn_2)$. According to the construction of pst'_Σ, it holds that

$$rgn_3 \in pst_\Sigma(rgn_1) \quad \wedge \quad rgn_3 \cap ngt(rgn_1) = \emptyset \quad \text{and}$$
$$rgn_3 \in pst_\Sigma(rgn_2) \quad \wedge \quad rgn_3 \cap ngt(rgn_2) = \emptyset.$$

Then, according to the construction of pst_Σ,

$$rgn_3 \in \coprod pst_\Pi(rgn_1) \quad \wedge \quad rgn_3 \cap ngt(rgn_1) = \emptyset \quad \text{and}$$
$$rgn_3 \in \coprod pst_\Pi(rgn_2) \quad \wedge \quad rgn_3 \cap ngt(rgn_2) = \emptyset.$$

Following the construction of pst_Π:

$$(\forall r \in rgn_3 \exists P \in pst_\Pi(rgn_1) \colon r \in P) \wedge (\forall P' \in pst_\Pi(rgn_1) \exists r' \in rgn_3 \colon r' \in P') \wedge$$
$$(rgn_3 \cap ngt(rgn_1) = \emptyset) \wedge$$
$$(\forall r \in rgn_3 \exists P \in pst_\Pi(rgn_2) \colon r \in P) \wedge (\forall P' \in pst_\Pi(rgn_2) \exists r' \in rgn_3 \colon r' \in P') \wedge$$
$$(rgn_3 \cap ngt(rgn_2) = \emptyset).$$

According to the construction of pst_Π, it holds that if $P \in pst_\Pi(rgn)$ then there exists a register $r \in rgn$ such that $P = up(r_i)$. Therefore, for each $P \in pst_\Pi(rgn)$ there exists a register $r' \in rgn'$ such that $r \in P$. Then, continuing in the proof, we obtain

$$(\forall r \in rgn_3 \exists up(r_i) \colon r_i \in rgn_1 \wedge r \in up(r_i)) \wedge (\forall r^\bullet \in rgn_1 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)) \wedge$$
$$(rgn_3 \cap ngt(rgn_1) = \emptyset) \wedge$$
$$(\forall r \in rgn_3 \exists up(r_i) \colon r_i \in rgn_2 \wedge r \in up(r_i)) \wedge (\forall r^\bullet \in rgn_2 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)) \wedge$$
$$(rgn_3 \cap ngt(rgn_2) = \emptyset)$$

From the construction of $ngt(rgn)$, the formula $rgn_i \cap ngt(rgn_j) = \emptyset$ is equivalent to the formula $\forall r \in rgn_i \neg\exists r^* \notin rgn_j \colon r \in up(r^*)$. Therefore:

$$\begin{gathered}(\forall r \in rgn_3 \exists up(r_i) \colon r_i \in rgn_1 \wedge r \in up(r_i)) \wedge (\forall r^\bullet \in rgn_1 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)) \wedge \\ (\forall r \in rgn_3 \neg\exists r^* \notin rgn_1 \colon r \in up(r^*)) \wedge \\ (\forall r \in rgn_3 \exists up(r_i) \colon r_i \in rgn_2 \wedge r \in up(r_i)) \wedge (\forall r^\bullet \in rgn_2 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)) \wedge \\ (\forall r \in rgn_3 \neg\exists r^* \notin rgn_2 \colon r \in up(r^*))\end{gathered}$$

Further, we only make use of

$$(\forall r \in rgn_3 \neg\exists r^* \notin rgn_1 \colon r \in up(r^*)) \quad \wedge \quad (\forall r^\bullet \in rgn_2 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)),$$

and the fact that rgn_1 and rgn_2 are distinct. Therefore, $\exists r^{dist} \colon r^{dist} \in rgn_2 \wedge r^{dist} \notin rgn_1$ (or vice versa).

By simplifying

$$\begin{gathered}(\exists r^{dist} \colon r^{dist} \in rgn_2 \wedge r^{dist} \notin rgn_1) \wedge \\ (\forall r \in rgn_3 \neg\exists r^* \notin rgn_1 \colon r \in up(r^*)) \wedge \\ (\forall r^\bullet \in rgn_2 \exists r' \in rgn_3 \colon r' \in up(r^\bullet)),\end{gathered}$$

we obtain

$$\begin{gathered}(\exists r^{dist} \colon r^{dist} \in rgn_2 \wedge r^{dist} \notin rgn_1) \wedge \\ (\forall r \in rgn_3 \forall r^* \colon r^* \notin rgn_1 \rightarrow r \notin up(r^*)) \wedge \\ (\exists r' \in rgn_3 \colon r' \in up(r^{dist})),\end{gathered}$$

which is clearly a contradiction, since $r^{dist} \notin rgn_1 \wedge \exists r' \in rgn_3 \colon r' \in up(r^{dist})$. $\lhd$

Finally, the marking M_F to be covered is constructed as $M_F = \mathbf{u}_{fin}$. We have finished the construction of $\mathcal{N}_\mathcal{A}$, now we need to show that it preserves the answer.

Claim 4.3.2. *$\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff the marking M_F is coverable in $\mathcal{N}_\mathcal{A}$.*

Proof. ($\Rightarrow$) Let $w \in (\Sigma \times \mathbb{D})^*$ such that $w \in \mathcal{L}(\mathcal{A})$. Moreover, assume that

$$\rho \colon c_0 \vdash_{t_1}^{w_1} c_1 \vdash_{t_2}^{w_2} \ldots \vdash_{t_n}^{w_n} c_n$$

is an accepting run of $\mathcal{A}$ on w. We will show that there exists a sequence of firings

$$\rho' \colon M_{init}[t'_{init}\rangle \;\; M_0[t'_1\rangle M_1[t'_2\rangle \ldots [t'_n\rangle M_n \;\; [t'_{fin}\rangle M_{fin}$$

in $\mathcal{N}_\mathcal{A}$ such that M_{fin} covers M_F. In particular, we construct the markings and transitions as follows:

- $M_{init} = \mathbf{u}_{init}$ and $t'_{init} = \langle \mathbf{u}_{init}, \mathbf{u}_{q_0}, \mathbf{id}\rangle$ for $c_0 = (q_0, f_0)$.
- For all $0 \leq i \leq n$ with $c_i = (q_i, f_i)$, we set M_i as follows:

$$\begin{gathered}M_i = \{init \mapsto 0, fin \mapsto 0, q_i \mapsto 1\} \cup \{q \mapsto 0 \mid q \in Q \setminus \{q_i\}\} \cup \\ \Big\{rgn \mapsto x \mid rgn \subseteq \mathbf{R}, x = \big|\cap_{r \in rgn} f_i(r)\big|\Big\}\end{gathered}$$

  Furthermore, $t'_i = \gamma(t_i, rgn_g)$ where $rgn_g = \{r \mid d_i \in f_{i-1}(r)\}$.

- M_{fin} is as follows:

$$M_{fin} = \{init \mapsto 0, fin \mapsto 1\} \cup \{q \mapsto 0 \mid q \in Q\} \cup \{rgn \mapsto M_n(rgn) \mid rgn \subseteq \mathbf{R}\}$$

and $t'_{fin} = \langle \mathbf{u}_{q_n}, \mathbf{u}_{fin}, \mathbf{id} \rangle$ for $c_n = (q_n, f_n)$.

Note that M_F is covered by M_{fin}.

We can now show by induction that ρ' is valid, i.e., all firings are enabled and respect the transition relation.

($\Leftarrow$) Let $\rho\colon M_{init}[t_0\rangle\ M_0[t_1\rangle M_1[t_2\rangle \dots [t_n\rangle M_n\ [t_{fin}\rangle M_{fin}$ be a run of $\mathcal{N}_\mathcal{A}$ such that $M_{fin} \geq M_F$, where M_F is the final marking. We show that there exists a sequence of transitions

$$\rho'\colon c_0 \vdash^{w_1}_{t'_1} c_1 \vdash^{w_2}_{t'_2} \dots \vdash^{w_n}_{t'_n} c_n$$

in $\mathcal{A}$ on $w = w_1 w_2 \dots w_n$, such that c_n is a final configuration, and, therefore, $w \in \mathcal{L}(\mathcal{A})$. First, we notice the following easy-to-see invariant of $\mathcal{N}_\mathcal{A}$, which holds for every M_i:

$$\sum_{p \in Q \cup \{init, fin\}} M_i(p) = 1 \tag{4.7}$$

i.e., there is always exactly one token in any of the places in $Q \cup \{init, fin\}$.

Let us now construct ρ' as follows:

- $c_0 = (q_0, f_0)$ is constructed such that q_0 is picked to be the state $q_0 \in Q$ with $M_0(q_0) = 1$ (this is well defined due to Equation (4.7)).
- For all $1 \leq i \leq n$, the transition t'_i is picked to be the transition such that $t_i \in \gamma(t'_i, rgn_g)$ for some region rgn_g. The data value d_i of w_i is then chosen to be compatible with the guard of rgn_g, i.e., $d_i \in \cap_{r \in rgn_g} f_{i-1}(r)$ and $d_i \notin \cup_{r \in \mathbf{R} \setminus rgn_g} f_{i-1}(r)$.

It can then be shown by induction that, for all $0 \leq i < n$, the following holds:

(a) $c_i = (q_i, f_i)$ where q_i is the (exactly one) state such that $M_i(q_i) = 1$ and

(b) the transition t_{i+1} is enabled.

We can then conclude that, since the last firing in ρ was $M_n[t_{fin}\rangle M_{fin}$, then, from the construction of $\mathcal{N}_\mathcal{A}$, it holds that $M_n(q_f) = 1$ for $q_f \in F$ and so ρ' is accepting. ◁

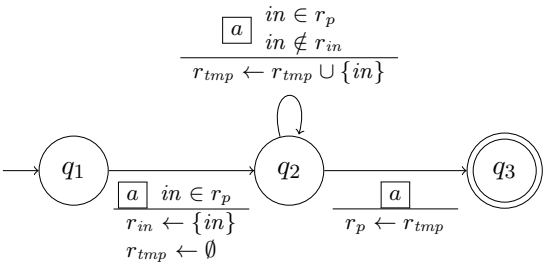
Claim 4.3.2 and the observation that $\mathcal{N}_\mathcal{A}$ is single-exponentially larger than $\mathcal{A}$ conclude the proof ($\mathbf{F}_\omega$ is closed under primitive-recursive reductions).

4.3.2 Reduction from coverability in TPN to emptiness in RsA

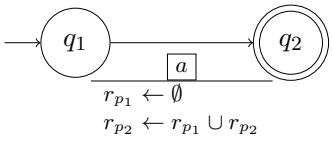
In the following, we give the formal definition of the reduction from coverability in transfer Petri nets to emptiness testing in register set automata. The reduction is followed by a proof that shows that it preserves the answer of the reduced problem.

Lemma 4.3.2. *The emptiness problem for* RsA *is* $\mathbf{F}_\omega$*-hard.*

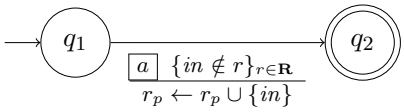
Proof. The proof is based on a reduction of coverability in TPNs (which is $\mathbf{F}_\omega$-complete) to non-emptiness of RsAs. Intuitively, given a TPN $\mathcal{N}$, we will construct the RsA $\mathcal{A}_\mathcal{N}$ simulating $\mathcal{N}$, which will have the following structure:



(a) The LossyRm(p) protogadget.



(b) The Move(p_1, p_2) protogadget.



(c) The NewToken(p) protogadget.

Figure 4.5: Protogadgets used in the construction of $\mathrm{RsA}_{\mathcal{N}}$.

- There will be the state q_{main}, which will be active before and after simulating the firing of TPN transitions.
- Each place of $\mathcal{N}$ will be simulated by a register of $\mathcal{A}_{\mathcal{N}}$; every token of $\mathcal{N}$ will be simulated by a unique data value.
- For every TPN transition, $\mathcal{A}_{\mathcal{N}}$ will contain a *gadget* that transfers data values between the registers representing the places active in the TPN transition. The gadget will start in q_{main} and end also in q_{main}.
- Coverability of a marking will be simulated by another gadget connected to q_{main} that will try to remove the number of tokens given in the marking from the respective places and arrive at the single final state q_{fin}.

Formally, let $\mathcal{N} = (P, T, M_0)$ with $P = \{p_1, \ldots, p_n\}$ be a TPN. W.l.o.g. we can assume that M_0 contains a single token in the place p_1, i.e., $M_0 = \{p_1 \mapsto 1, p_2 \mapsto 0, \ldots, p_n \mapsto 0\}$. We will show how to construct the RsA $\mathcal{A}_{\mathcal{N}} = (Q, \mathbf{R}, \Delta, \{q_{init}\}, \{q_{fin}\})$ over the unary alphabet $\Sigma = \{a\}$ such that a marking M_f is coverable in $\mathcal{N}$ iff the language of $\mathcal{A}_{\mathcal{N}}$ is non-empty. The set of registers of $\mathcal{A}_{\mathcal{N}}$ will be the set $\mathbf{R} = \{r_{in}, r_{tmp}\} \cup \{r_p, r_{p'} \mid p \in P\}$.

Protogadgets. Let us now define the following *protogadgets*, which we will later use for creating a *gadget* for each TPN transition and a gadget for doing the coverability test. We define the following protogadgets:

1. The *Lossy Removal* protogadget, which simulates a (lossy) removal of one token from a place p is the RsA defined as $\textsc{LossyRm}(p) = (\{q_1, q_2, q_3\}, \mathbf{R}, \Delta', \{q_1\}, \{q_3\})$ where Δ' contains the following three transitions (cf. Figure 4.5a):

$$\Delta' = \left\{ \begin{array}{c} q_1 \xrightarrow{a \mid \{r_p\}, \emptyset, \{r_{in} \mapsto \{in\}, r_{tmp} \mapsto \emptyset\}} q_2, \\ q_2 \xrightarrow{a \mid \{r_p\}, \{r_{in}\}, \{r_{tmp} \mapsto \{r_{tmp}, in\}\}} q_2, \\ q_2 \xrightarrow{a \mid \emptyset, \emptyset, \{r_p \mapsto \{r_{tmp}\}\}} q_3 \end{array} \right\} \tag{4.8}$$

Intuitively, the protogadget stores the data value to be removed from p in a special register r_{in}. Next, it simulates the calculation of the difference of r_p and r_{in}. This is done by accumulating the values that are present in r_p and are not present in r_{in} into r_{tmp}.

Since some values may get "lost", and disappear because of not being added to the accumulated difference, this protogadget is considered *lossy*.

2. The *Move* protogadget, which simulates *moving* all tokens from a place p_1 to a place p_2, is the following RsA (also depicted in Figure 4.5b):

$$\textsc{Move}(p_1, p_2) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \emptyset, \{r_{p_1} \mapsto \emptyset, r_{p_2} \mapsto \{r_{p_1}, r_{p_2}\}\}} q_2\}, \{q_1\}, \{q_2\}).$$

Intuitively, the protogadet empties out the register that represents the place p_1. Its previous value is assigned to the register representing the place p_2 in union with its value.

3. The *New Token* protogadget, which simulates adding a token to a place p, is defined as follows (depiction is in Figure 4.5c):

$$\textsc{NewToken}(p) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \mathbf{R}, \{r_p \mapsto \{r_p, in\}\}} q_2\}, \{q_1\}, \{q_2\}).$$

Intuitively, the protogadget adds the unique data value from the input tape into the register representing the place p. The uniqueness of the data value is ensured by $g^{\notin}$, which requires that the input in does not belong to any of the registers from $\mathbf{R}$.

For convenience, we will use the following notation. Let $\mathcal{A}_1 = (Q_1, \mathbf{R}, \Delta_1, \{q_1^I\}, \{q_1^F\})$ and $\mathcal{A}_2 = (Q_2, \mathbf{R}, \Delta_2, \{q_2^I\}, \{q_2^F\})$ be a pair of RsAs with a single initial state and a single final state. We will use $\mathcal{A}_1 \cdot \mathcal{A}_2$ to denote the RsA

$$(Q_1 \uplus Q_2, \mathbf{R}, \Delta_1 \cup \{q_1^F \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q_2^I\} \cup \Delta_2, \{q_1^I\}, \{q_2^F\}).$$

Moreover, for $n \in \mathbb{N}_0$, we use $\mathcal{A}_1^{[n]}$ to denote the RsA defined inductively as

$$\begin{aligned} \mathcal{A}_1^{[0]} &= (\{q\}, \mathbf{R}, \emptyset, \{q\}, \{q\}), \\ \mathcal{A}_1^{[i+1]} &= \mathcal{A}_1^{[i]} \cdot \mathcal{A}_1. \end{aligned}$$

Intuitively, $\mathcal{A}_1^{[n]}$ is a concatenation of n copies of $\mathcal{A}_1$.

Gadgets. For each TPN transition $t = \langle In, Out, Transfer \rangle$, we then create the *gadget* RsA $\mathcal{A}_t$ in several steps.

1. First, we transform In into the RsA $\mathcal{A}_{In} = \mathcal{A}_{In(p_1)} \cdot \ldots \cdot \mathcal{A}_{In(p_n)}$ where every $\mathcal{A}_{In(p_i)}$ is defined as $\mathcal{A}_{In(p_i)} = \textsc{LossyRm}(p_i)^{[In(p_i)]}$, i.e., it is a concatenation of $In(p_i)$ copies of $\textsc{LossyRm}(p_i)$.

2. Second, from Out we create the RsA $\mathcal{A}_{Out} = \mathcal{A}_{Out(p_1)} \cdot \ldots \cdot \mathcal{A}_{Out(p_n)}$ with $\mathcal{A}_{Out(p_i)}$ defined as $\mathcal{A}_{Out(p_i)} = \textsc{NewToken}(p_i)^{[Out(p_i)]}$, i.e., it is a concatenation of $Out(p_i)$ copies of $\textsc{NewToken}(p_i)$.

3. Third, from *Transfer* we obtain the RsA $\mathcal{A}_{Transfer} = \mathcal{A}_{Transfer(p_1)} \cdot \ldots \cdot \mathcal{A}_{Transfer(p_n)} \cdot \mathcal{A}_{unprime(p_1)} \cdot \ldots \cdot \mathcal{A}_{unprime(p_n)}$ such that $\mathcal{A}_{Transfer(p_i)} = \textsc{Move}(r_{p_i}, r_{p'_j})$ with $p_j = Transfer(p_i)$ and $\mathcal{A}_{unprime(p_i)} = \textsc{Move}(p'_i, p_i)$. Intuitively, $\mathcal{A}_{Transfer}$ first moves the contents of all registers according to *Transfer* to primed instances of the target registers (in order to avoid mix-up) and then unprimes the register names.

4. Finally, we combine the RsAs created above into the single gadget obtained as $\mathcal{A}_t = \mathcal{A}_{In} \cdot \mathcal{A}_{Transfer} \cdot \mathcal{A}_{Out}$.

The initial marking will be encoded by a gadget that puts one new data value in the register representing the place p_1. For this, we construct the RsA $\mathcal{A}_{M_0} = \textsc{NewToken}(p_1)$ and rename its initial state to q_{init}.

The last ingredient we need is to create a gadget that will encode the marking M_f, whose coverability we are checking. For this, we construct the gadget $\mathcal{A}_{M_f} = \mathcal{A}_{M_f(p_1)} \cdot \ldots \cdot \mathcal{A}_{M_f(p_n)}$ where every $\mathcal{A}_{M_f(p_i)}$ is defined as $\mathcal{A}_{M_f(p_i)} = \textsc{LossyRm}(p_i)^{[M_f(p_i)]}$, i.e., it is a concatenation of $M_f(p_i)$ copies of $\textsc{LossyRm}(p_i)$. We rename the final state of $\mathcal{A}_{M_f}$ to q_{fin}. W.l.o.g. we assume that the set of states of all constructed gadgets are pairwise disjoint.

Construction of $\mathcal{A}_\mathcal{N}$. We can now finalize the construction. $\mathcal{A}_\mathcal{N}$ is obtained as the union of the following RsAs:

- $\mathcal{A}_{M_0} = (Q_{M_0}, \mathbf{R}, \Delta_{M_0}, \{q_{init}\}, \{q^F_{M_0}\})$,
- $\mathcal{A}_{M_f} = (Q_{M_f}, \mathbf{R}, \Delta_{M_f}, \{q^I_{M_f}\}, \{q_{fin}\})$, and
- $\mathcal{A}_t = (Q_t, \mathbf{R}, \Delta_t, \{q^I_t\}, \{q^F_t\})$ for every $t \in T$.

Each of the previous RsAs is connected to the state q_{main}. The final, reduced register set automaton is defined as, $\mathcal{A}_\mathcal{N} = (Q, \mathbf{R}, \Delta, \{q_{init}\}, \{q_{fin}\})$, where

- $Q = \{q_{main}\} \cup Q_{M_0} \cup Q_{M_f} \cup \bigcup_{t \in T} Q_t$ and
- $\Delta = \Delta_{M_0} \cup \Delta_{M_f} \cup \{q^F_{M_0} \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q_{main}, q_{main} \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q^I_{M_f}\} \cup \bigcup_{t \in T} (\Delta_t \cup \{q_{main} \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q^I_t, q^F_t \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q_{main}\})$.

In the following, we prove that reduction to the emptiness preserves the answer of the coverability.

Claim 4.3.3. *The marking M_F is coverable in $\mathcal{N}$ iff $\mathcal{L}(\mathcal{A}_\mathcal{N}) \neq \emptyset$.*

Proof. ($\Rightarrow$) Let there be the following run of $\mathcal{N}$:

$$\rho\colon M_0[t_1\rangle M_1[t_2\rangle \ldots [t_n\rangle M_n$$

such that M_n covers M_F. We will show that there exists a word $w \in (\Sigma \times \mathbb{D})^*$ and a run

$$\begin{aligned}
\rho'\colon\ & c_{(init,0)} \vdash^{w_1}_{t'_{(init,1)}} c_{(init,1)} \vdash^{w_2}_{t'_{(init,2)}} \ldots c_{(init,k_{init})} \vdash^{w_{i_0}}_{t'_{(0,0)}} && \text{(initialization)}\\
& c_{(0,0)} \vdash^{w_{i_0+1}}_{t'_{(0,1)}} c_{(0,1)} \ldots c_{(0,k_1)} \vdash^{w_{i_1}}_{t'_{(1,0)}} c_{(1,0)} \vdash^{w_{i_1+1}}_{t'_{(1,1)}} c_{(1,1)} \ldots c_{(1,k_1)} \ldots c_{(n-1,k_{n-1})} \vdash^{w_{i_n}}_{t'_{(n,0)}}\\
& c_{(n,0)} \vdash^{w_{i_n+1}}_{t'_{(fin,1)}} c_{(fin,1)} \ldots \vdash^{w_{i_{fin}}}_{t'_{(fin,k_{fin})}} c_{(fin,k_{fin})} && \text{(finalization)}
\end{aligned}$$

of $\mathcal{A}_\mathcal{N}$ on w such that $q \in F$ for $c_{(fin,k_{fin})} = (q, \cdot)$. The run ρ' will be constructed to preserve the following invariant for each $0 \leq i \leq n$:

$$c_{(i,0)} = (q_{main}, f_i) \quad \text{such that} \quad \forall p \in P\colon |f_i(r_p)| = M_i(p). \tag{4.9}$$

Therefore, configurations with state q_{main} represent the TPN's state after (or before) firing a transition. Firing a transition t is simulated by going to the gadget for t in $\mathcal{A}_\mathcal{N}$ and picking input data values such that the run returns to q_{main} in as many steps as possible (this is to take the run through the LossyRm protogadgets that preserves the precise value of the marking). By induction on $0 \leq i \leq n$, we can show that the invariant in Equation (4.9) is preserved (the base case is proved by observing that the *initialization* part is correct).

($\Leftarrow$) Let $w \in \mathcal{L}(\mathcal{A}_\mathcal{N})$ and

$$\begin{aligned}
\rho\colon\ & c_{(init,0)} \vdash^{w_1}_{t'_{(init,1)}} c_{(init,1)} \vdash^{w_2}_{t'_{(init,2)}} \dots c_{(init,k_{init})} \vdash^{w_{i_0}}_{t'_{(0,0)}} && \text{(initialization)}\\
& c_{(0,0)} \vdash^{w_{i_0+1}}_{t'_{(0,1)}} c_{(0,1)} \dots c_{(0,k_1)} \vdash^{w_{i_1}}_{t'_{(1,0)}} c_{(1,0)} \vdash^{w_{i_1+1}}_{t'_{(1,1)}} c_{(1,1)} \dots c_{(1,k_1)} \dots c_{(n-1,k_{n-1})} \vdash^{w_{i_n}}_{t'_{(n,0)}} \\
& c_{(n,0)} \vdash^{w_{i_n+1}}_{t'_{(fin,1)}} c_{(fin,1)} \dots \vdash^{w_{i_{fin}}}_{t'_{(fin,k_{fin})}} c_{(fin,k_{fin})} && \text{(finalization)}
\end{aligned}$$

be an accepting run of $\mathcal{A}_\mathcal{N}$ on w such that for each $0 \leq i \leq n$, it holds that $c_{(i,0)} = (q_{main}, \cdot)$—this follows from the structure of $\mathcal{A}_\mathcal{N}$. We will construct a run

$$\rho'\colon M_0[t_1\rangle M_1[t_2\rangle \dots [t_n\rangle M_n$$

where each t_i is the TPN's transition corresponding to the gadget that the corresponding part of ρ traversed. For all $0 \leq i \leq n$, the following invariant will hold:

$$\forall p \in P\colon |f_i(r_p)| \leq M_i(p). \tag{4.10}$$

We note that the run ρ might not have been the "*most precise*" run of $\mathcal{A}_\mathcal{N}$, so the markings in the TPN run overapproximate the contents of $\mathcal{A}_\mathcal{N}$'s registers in ρ. The invariant can be proved by induction. ◁

Claim 4.3.3 and the observation that $\mathcal{A}_\mathcal{N}$ is single-exponentially larger than $\mathcal{N}$ (assuming binary encoding of the numbers in $\mathcal{N}$) conclude the proof ($\mathbf{F}_\omega$ is closed under primitive-recursive reductions). □

From Lemma 4.3.1 and Lemma 4.3.2, we immediately obtain Theorem 4.3.1.

Chapter 5

Determinisation of Register Automata

Determinisation is a process of transforming a formal model into one whose each step of any run is deterministic, i.e., there is always at most one possibility of moving into the next state with the current input, and the languages of the two models are equivalent. In other words, determinisation is a process of converting a non-deterministic automaton into a deterministic one. Amongst other things, the process of converting automata to their equivalent, deterministic versions is a crucial step for increasing the efficiency of some automata-based approaches since it eliminates the need of performing backtracking in some practical applications. Generally speaking, some operations, such as complementation, are much easier in case of deterministic automata. In addition, considering a deterministic model of register automata allows for regaining the decidability of some decision problems, including language inclusion, equivalence, and universality.

Register automata themselves cannot always be determinised. One of the main motivations for this thesis was the introduction of a register automaton model that can possess the quality of deterministic computation. Even though some previous advances in the theory of register automata have shown that a data language is recognizable by a DRA if, and only if, both this language and its complement are recognizable by NRAs [24], we wish to extend the property of determinisability onto a bigger class of register automata. RsAs do have this interesting property: a large class of NRA languages can, indeed, be determinised into DRsAs. In this chapter, we give a semi-algorithm for determinisation of register automata into register set automata, and we discuss properties of a class of NRAs for which the given algorithm is complete.

Preprocessing. Let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be an NRA. We use $\mathbf{R}[q]$ for $q \in Q$ to denote the set of registers r such that there exists a transition $s \xrightarrow{\cdot \mid g^{=}, g^{\neq}, up} t \in \Delta$ with

(i) $up(r) \neq \bot$ and $t = q$ or
(ii) $r \in g^{=} \cup g^{\neq}$ and $s = q$.

Intuitively, $\mathbf{R}[q]$ denotes the set of registers *active* in q. Given a set of states S, we define

$$\mathbf{R}[S] = \bigcup_{q \in S} \mathbf{R}[q].$$

We call $\mathcal{A}$ *register-local* if for all $r \in \mathbf{R}$ it holds that if $r \in \mathbf{R}[q]$ and $r \in \mathbf{R}[s]$ for some states $q, s \in Q$, then $q = s$. It is easy to see that every NRA can be transformed into

the register-local form by creating a new copy of a register for every state that uses it, potentially increasing the number of registers to $|Q| \cdot |\mathbf{R}|$.

Furthermore, we call $\mathcal{A}$ *single-valued* if there is no reachable configuration (q, f) such that $f(r_1) = f(r_2)$ for a pair of distinct registers $r_1, r_2 \in \mathbf{R}$, i.e., there is at most one copy of each data value in $\mathcal{A}$. Again, any NRA can be converted into the single-valued form, however, the number of states can increase to $B_{|\mathbf{R}|} \cdot |Q|$ where B_n is the n-th Bell number[1]. Intuitively, the transformation is done by creating one copy of each state for every possible partition of $\mathbf{R}$ (the partitions denote which registers hold the same value), and modifying the transition function correspondingly.

The algorithm. The determinisation (semi-)algorithm for a single-valued NRA $\mathcal{A}$ is shown in Algorithm 1. On the high level, it is similar to the standard Rabin-Scott subset construction for determinising finite automata [30] with additional treatment of registers superimposed onto it.[2]

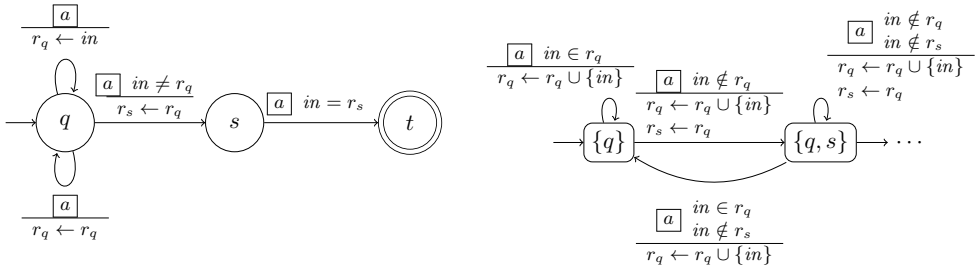
Algorithm 1: Determinisation of an NRA into a DRsA

Input : Single-valued NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$
Output: DRsA $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', F')$ with $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ or $\bot$

$\mathcal{Q}' \leftarrow$ worklist $\leftarrow I' \leftarrow \{(I, c_0 = \{r \mapsto 0 \mid r \in \mathbf{R}\})\}$;
$\Delta' \leftarrow \emptyset$;
while worklist $\neq \emptyset$ **do**
  $(S, c) \leftarrow$ worklist.$pop()$;
  foreach $a \in \Sigma, g \subseteq \mathbf{R}$ **do**
    $T \leftarrow \{q \xrightarrow{a \mid g^{=}, g^{\neq}, \cdot} q' \in \Delta \mid q \in S, g^{=} \subseteq g, g^{\neq} \cap g = \emptyset\}$;
    $S' \leftarrow \{q' \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, \cdot} q' \in T\}$;
    if $\exists q \xrightarrow{\cdot \mid \cdot, g^{\neq}, \cdot} q' \in T, \exists r \in g^{\neq} : c(r) = \omega$ **then return** $\bot$;
    foreach $r_i \in \mathbf{R}$ **do**
      $tmp \leftarrow \{x \in \mathbf{R} \cup \{in\} \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} \cdot \in T, up(r_i) = x, c(r_i) \neq 0\}$;
      $op_{r_i} \leftarrow (tmp \setminus g) \cup \{in \mid x \in tmp \cap g\}$;
    foreach $q' \in S'$ **do**
      $P \leftarrow op_{r_1} \times \cdots \times op_{r_n}$ for $\{r_1, \ldots, r_n\} = \mathbf{R}[q']$;
      foreach $(x_1, \ldots, x_n) \in P$ **do**
        if $\nexists (\cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} q') \in T$ *s.t.* $\bigwedge_{1 \le i \le n} up(r_i) = x_i$ **then return** $\bot$;
    $up' \leftarrow \{r_i \mapsto op_{r_i} \mid r_i \in \mathbf{R}\}$;
    $c' \leftarrow \{r_i \mapsto \sum_{x \in up'(r_i)} \hat{c}(x, g) \mid r_i \in \mathbf{R}\}$;
    if $(S', c') \notin \mathcal{Q}'$ **then**
      worklist.$push((S', c'))$;
      $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{(S', c')\}$;
    $\Delta' \leftarrow \Delta' \cup \{(S, c) \xrightarrow{a \mid g, \mathbf{R} \setminus g, up'} (S', c')\}$;
return $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', \{(S, c) \in \mathcal{Q}' \mid S \cap F \neq \emptyset\})$;

[1] Bell number represents the number of possible partitions of a given set, where partition stands for grouping of set's elements into non-empty subsets, such that these subsets are pairwise disjoint

[2] The algorithm can be seen as a simplification of the algorithm for converting counting automata to deterministic counting-set automata from [44] (we do not need to deal with operations on the values stored in registers), but with additional features necessary to deal with NRA-specific issues.



(a) An input NRA $\mathcal{A}$ with a disequality guard.

(b) A part of the RsA obtained for $\mathcal{A}$.

Figure 5.1: Possible inconsistency in determinisation caused by disequality on guards in the input automaton.

During the construction, we track

(i) all states of $\mathcal{A}$ in which the runs of $\mathcal{A}$ might be at a given point, represented by a set of states $S \subseteq Q$ and

(ii) the sizes of sets stored in each register, represented by a mapping $c\colon \mathbf{R} \to \{0, 1, \omega\}$ (ω denotes any number ≥ 2);

the macrostate is then a pair (S, c). Regarding the c-element of the macrostate, we keep track of the sizes to detect when our simulation of a disequality test $in \neq r$ by the non-membership test $in \notin r$ is imprecise due to r containing two or more elements. The initial state of the constructed DRsA is the macrostate (I, c_0) where c_0 is a mapping assigning zero to each register (the run of a DRsA starts with all registers initialized to $\emptyset$) (Line 1).

The main loop of the algorithm then constructs successors of reachable macrostates for each $a \in \Sigma$ and each $g \subseteq \mathbf{R}$ on Line 5; each pair a, g corresponds to the so-called *minterm* (minterms denote combinations of guards whose semantics do not overlap [9]). For each minterm, we collect all transitions of $\mathcal{A}$ compatible with this minterm (Line 6) and generate the successor set of states S' (Line 7). The $\mathcal{A}'$ update function up' for register r is then set to collect into r all possible values that might be stored into r in $\mathcal{A}$ on any run over the input word at the given position (Lines 9–16).

The algorithm needs to avoid the following possible issues:

1. Since the algorithm collects in the set-register r all possible values that could have been stored into the standard register r in $\mathcal{A}$, if the disequality tests in $g^{\neq}$ were changed for non-membership tests in $g^{\notin}$, this could mean that $\mathcal{A}'$ might not be able to simulate some transition of $\mathcal{A}$ (the transition would not be enabled). Consider the example in Figure 5.1, where Figure 5.1b contains a part of the RsA obtained if Algorithm 1 did not use the c-component of macrostates. Notice that while $\mathcal{A}$ does accept the word $\langle a, 1\rangle\langle a, 2\rangle\langle a, 2\rangle\langle a, 1\rangle$, the RsA obtained in this way does not. The reason for this is that after reading the third symbol (i.e., $\langle a, 2\rangle$), the RsA goes to the macrostate $\{q\}$—it thinks it cannot be in s any more.

   This is the reason why we augment macrostates with the c-component. If we detect that a disequality test is performed on a register containing more than one element, we terminate the algorithm (Line 8). The tracking of sizes of sets stored in registers is done on Line 17 where $\hat{c}(x, g)$ is defined as

(i) $c(x)$ if $x \in \mathbf{R} \setminus g$,

(ii) 0 if $x \in g$, and

(iii) 1 if $x = in$;

moreover, the sum is *saturated* to ω for values ≥ 2.

2. By collecting all possible values that can occur in registers, the algorithm is performing the so-called *Cartesian abstraction* (i.e., it is losing information about dependencies between components in tuples). This can lead to a scenario where, for some set-register assignment f' of $\mathcal{A}'$, we would have $d_1 \in f'(r_1)$ and $d_2 \in f'(r_2)$, but there would be no corresponding configuration of $\mathcal{A}$ with register assignment f such that $d_1 = f(r_1)$ and $d_2 = f(r_2)$. Consider, e.g., an NRA for the language $\{uvwvz \mid u, w, z \in (\Sigma \times \mathbb{D})^*, |v| = 2\}$ depicted in Figure 5.2. When the algorithm computes the successor of the macrostate $(\{q, s, t\}, \{r_1{:}1, r_2{:}1, r_3{:}1\})$ over $a \in \Sigma$ and the guard $g = \emptyset$, it would obtain the following update of registers: $r_1 \leftarrow \{in\}$ (transition from q to s), $r_2 \leftarrow r_1 \cup r_2$ (transition from s to t and transition from t to t), and $r_3 \leftarrow r_3 \cup \{in\}$ (transition from s to t and transition from t to t). This would simulate the update $r_2 \leftarrow r_2$, $r_3 \leftarrow in$, which is nowhere in the original NRA. The algorithm detects the possibility of such an overapproximation on Lines 12–15.

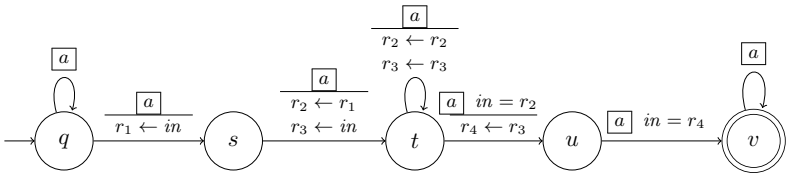
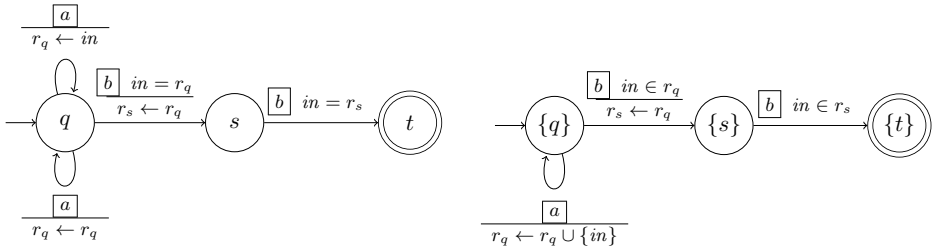


Figure 5.2: Example RsA.

3. We need to avoid a situation when a set-register has collected all possible nondeterministic choices of a standard NRA register and then is tested twice with a different result. Consider the example of an NRA $\mathcal{A}$ and an RsA obtained from $\mathcal{A}$ by Algorithm 1 without Line 11 (to save space, we collapse all macrostates with the same set of states into one) depicted in Figure 5.3. One can see that while the NRA cannot accept the word $\langle a, 1\rangle\langle a, 2\rangle\langle b, 1\rangle\langle b, 2\rangle$, the RsA accepts it. This happens because the RsA did not "*collapse*" the possible nondeterministic choices that are kept in the registers for the value of r_q after the first membership test (on the transition from $\{q\}$ to $\{s\}$) succeeded. We avoid this situation by the code on Line 11, which performs the collapse of the set of nondeterministic choices into a single value when it is positively tested. The update on the RsA transition from $\{q\}$ to $\{s\}$ constructed by the algorithm will then become $r_s \leftarrow \{in\}$ and the result will be precise.

   One might also imagine similar scenario as the previous but with several registers copying a nondeterministically chosen value (e.g., when a data value is copied from r_1 to r_2 and, later, r_1 is positively tested for equality, we need to guarantee that the value of r_2 also collapses to the given data value). In order to avoid this, we require that the input NRA is *single-valued*, i.e., it never happens that a data value is in more than one register.



(a) An NRA $\mathcal{A}$

(b) An RsA overapproximating $\mathcal{A}$'s language

Figure 5.3: Possible inconsistency in determinisation causing overapproximation of $\mathcal{A}$'s language.

Soundness of the algorithm. In the following, we prove that the determinisation preserves the language of the input NRA.

Theorem 5.0.1. *When Algorithm 1 returns a* DRsA $\mathcal{A}'$*, then* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Proof. ($\subseteq$) Let $w = \langle a_1, d_1\rangle \dots \langle a_n, d_n\rangle \in \mathcal{L}(\mathcal{A})$. Then there is an accepting run ρ of $\mathcal{A}$ on w, such that

$$\rho\colon (q_0, f_0) \vdash_{t_1}^{\langle a_1,d_1\rangle} (q_1, f_1) \vdash_{t_2}^{\langle a_2,d_2\rangle} \cdots \vdash_{t_n}^{\langle a_n,d_n\rangle} (q_n, f_n)$$

with $q_n \in F$. Furthermore, let

$$\rho'\colon ((S'_0, c'_0), f'_0) \vdash_{t'_1}^{\langle a_1,d_1\rangle} ((S'_1, c'_1), f'_1) \vdash_{t'_2}^{\langle a_2,d_2\rangle} \cdots \vdash_{t'_n}^{\langle a_n,d_n\rangle} ((S'_n, c'_n), f'_n)$$

be the run of $\mathcal{A}'$ on w ($\mathcal{A}'$ is deterministic and complete, so ρ' is unique). We will show that ρ' is accepting, and so $w \in \mathcal{L}(\mathcal{A}')$.

Let us show that for all $0 \leq i \leq n$, the following conditions hold:

1. $q_i \in S'_i$,
2. $\forall r \in \mathbf{R}\colon f_i(r) \neq \bot \implies f_i(r) \in f'_i(r)$, and
3. $\forall r \in \mathbf{R}\colon c'_i(r) = \begin{cases} 0 & \text{iff } f'_i(r) = \emptyset, \\ 1 & \text{iff } |f'_i(r)| = 1, \\ \omega & \text{iff } |f'_i(r)| \geq 2. \end{cases}$

We proceed by induction on i:

- $i = 0$: Since the (only) initial state of $\mathcal{A}'$ is the macrostate $(I, c_0 = \{r_i \mapsto 0 \mid r_i \in \mathbf{R}\})$ (cf. Line 22), and $q_0 \in I$, then condition 1 holds. Moreover, $f_0(r) = \bot$ for every $r \in \mathbf{R}$, so condition 2 holds trivially, and so does condition 3 (the c'_i of all registers is initialised to zero on Line 1 and all registers are initialised to $\emptyset$ in a run of an RsA).
- $i = j + 1$: We assume the conditions hold for $i = j$. Let there be the following transition from the j-th to the $(j+1)$-th configurations of the runs ρ and ρ':

$$(q_j, f_j) \vdash_{t_{j+1}}^{\langle a_{j+1},d_{j+1}\rangle} (q_{j+1}, f_{j+1})$$

and

$$((S'_j, c'_j), f'_j) \vdash_{t'_{j+1}}^{\langle a_{j+1}, d_{j+1}\rangle} ((S'_{j+1}, c'_{j+1}), f'_{j+1})$$

and let $t_{j+1}\colon q_j \xrightarrow{a_{j+1} \mid g^{=}, g^{\neq}, up} q_{j+1}$. Moreover, let $g \subseteq \mathbf{R}$ be the set of registers r such that $f(r) = d_{j+1}$ (g here corresponds directly to the g on Line 5 of Algorithm 1). Transition t'_{j+1} would be $(S'_j, c'_j) \xrightarrow{a_{j+1} \mid g, \mathbf{R} \setminus g, up'} (S'_{j+1}, c'_{j+1})$ where S'_{j+1} is constructed using a_{j+1} and g as on Lines 6–7.

We need to show the following:

(i) transition $t'_{j+1}\colon (S'_j, c'_j) \xrightarrow{a_{j+1} \mid g, \mathbf{R} \setminus g, up'} (S'_{j+1}, c'_{j+1})$ is enabled and
(ii) conditions 1–3 hold for $i = j + 1$.

Claim 5.0.1. *Transition* $t'_{j+1}\colon (S'_j, c'_j) \xrightarrow{a_{j+1} \mid g, \mathbf{R} \setminus g, up'} (S'_{j+1}, c'_{j+1})$ *is enabled.*

Proof. Since transition t_{j+1} is enabled in configuration (q_j, f_j), it holds that

(a) $\forall r \in g^{=}\colon f_j(r) = d_{j+1}$ and
(b) $\forall r \in g^{\neq}\colon f_j(r) \neq d_{j+1}$.

From (a) and the induction hypothesis (condition 2), we have that $\forall r \in g\colon d_{j+1} \in f'_j(r)$, so the $g^{\in}$-part of t'_{j+1}'s enabledness holds. Proving the $g^{\notin}$-part (i.e., that $\forall r \in \mathbf{R} \setminus g\colon d_{j+1} \notin f'_j(r)$) is more difficult. We prove this by contradiction.

For the sake of contradiction, assume that there exists a register $r \in g^{\neq}$ such that $d_{j+1} \in f'_j(r)$ (other registers in $\mathbf{R} \setminus g$ do not need to be considered because they do not affect the enabledness of t_{j+1}). Because

(i) $f_j(r) \neq d_{j+1}$,
(ii) $f_j(r) \in f'_j(r)$ (from condition 2 of the induction hypothesis), and
(iii) $d_{j+1} \in f'_j(r)$ (from the assumption),

we know that $|f'_j(r)| \geq 2$. From condition 3 of the induction hypothesis, it holds that $c'_j(r) = \omega$. But then, since there is a register $r \in g^{\neq}$ such that $c'_j(r) = \omega$, Algorithm 1 would on Line 8 return $\bot$, which gives us a contradiction with the fact that the algorithm returned a DRsA. ◁

Claim 5.0.2. *The following holds:*

1. $q_{j+1} \in S'_{j+1}$,
2. $\forall r \in \mathbf{R}\colon f_{j+1}(r) \neq \bot \implies f_{j+1}(r) \in f'_{j+1}(r)$, *and*
3. $\forall r \in \mathbf{R}\colon c'_{j+1}(r) = \begin{cases} 0 & \text{iff } f'_{j+1}(r) = \emptyset, \\ 1 & \text{iff } |f'_{j+1}(r)| = 1, \\ \omega & \text{iff } |f'_{j+1}(r)| \geq 2. \end{cases}$

Proof. 1. Trivial.

2. Follows from the induction hypothesis and the definition of the update function up' on Lines 9–16.
3. Follows from the induction hypothesis and from the definition of c' on Line 17. ◁

This concludes the first direction of the proof.

($\supseteq$) Let $w = \langle a_1, d_1\rangle \dots \langle a_n, d_n\rangle \in \mathcal{L}(\mathcal{A}')$. Then there is an accepting run ρ' of $\mathcal{A}'$ on w, such that

$$\rho' \colon ((S'_0, c'_0), f'_0) \vdash_{t'_1}^{\langle a_1, d_1\rangle} ((S'_1, c'_1), f'_1) \vdash_{t'_2}^{\langle a_2, d_2\rangle} \cdots \vdash_{t'_n}^{\langle a_n, d_n\rangle} ((S'_n, c'_n), f'_n)$$

with $(S'_n, c'_n) \in F'$. We will construct in a backward manner a sequence of sets of $\mathcal{A}$'s configurations (i.e., pairs containing a state and assignment to registers) $U_0, U_1, \dots, U_{n-1}, U_n$, such that $\forall 1 \leq i \leq n \colon U_i \in Q \times (\mathbf{R} \to \mathbb{D})$, that represents all accepting runs of $\mathcal{A}$ over w, and show that U_0 contains a configuration (q_0, f_0) with $q_0 \in I$ and $f_0 = \{r \mapsto \bot \mid r \in \mathbf{R}\}$. Let us start with U_n, which we construct as $U_n = \{(q_n, f_n) \mid q_n \in S'_n \cap F', f_n = \{r \mapsto d \mid d \in f'_n(r)\}\}$. Now, given U_{i+1} for $0 \leq i \leq n-1$, we construct the set of previous configurations U_i as the set of pairs (q_i, f_i) for which the following conditions hold:

1. $q_i \in S'_i$,
2. for every register $r \in \mathbf{R}$, it holds that $f_i(r) \in f'_i(r) \cup \{\bot\}$,
3. there is a transition $t_{i+1} \colon q_i \xrightarrow{a_{i+1} \mid g^{=}, g^{\neq}, up} q_{i+1} \in \Delta$ such that
   * $(q_{i+1}, f_{i+1}) \in U_{i+1}$ and
   * $(q_i, f_i) \vdash_{t_{i+1}}^{\langle a_{i+1}, d_{i+1}\rangle} (q_{i+1}, f_{i+1})$.

Let us now show that for all $0 \leq i \leq n$, the set U_i is nonempty. We proceed by backward induction.

* $i = n$ (base case): ρ' is accepting, so there is at least one state in $S'_n \cap F'$.
* $i = j + 1$ (induction hypothesis): we assume that $U_{j+1} \neq \emptyset$.
* $i = j$ (induction step): because in configuration $((S'_j, c'_j), f'_j)$ the transition $t'_{j+1} \colon (S'_j, c'_j) \xrightarrow{a_{j+1} \mid g^{\in}, g^{\notin}, up'} (S'_{j+1}, c'_{j+1})$ is enabled, it needs to hold that
  - $\forall r \in g^{\in} \colon d_{j+1} \in f'_j(r)$ and
  - $\forall r \in g^{\notin} \colon d_{j+1} \notin f'_j(r)$.

  Then, for every q_{j+1} such that $(q_{j+1}, f_{j+1}) \in U_{j+1}$, from the construction of t'_{j+1}, there needs to exist a transition $t_{j+1} \colon q_j \xrightarrow{a_{j+1} \mid g^{=}, g^{\neq}, up} q_{j+1} \in \Delta$ with $g^{=} \subseteq g^{\in}$ and $g^{\neq} \subseteq g^{\notin}$. It is left to show that there is an assignment f_j such that $(q_j, f_j) \in U_j$ and $(q_j, f_j) \vdash_{t_{j+1}}^{\langle a_{j+1}, d_{j+1}\rangle} (q_{j+1}, f_{j+1})$. But this clearly holds since if it did not hold, then the algorithm would fail on Lines 13–15 (when checking whether the update is overapproximating or not) and would not produce $\mathcal{A}'$. □

Properties for determinisability. Naturally, we wish to syntactically characterize the class of NRAs for which Algorithm 1 is complete. We observe that when we start with an $\text{NRA}_1^{=}$ and transform it into the single-valued register-local form, the algorithm always returns a DRsA.

Theorem 5.0.2. *(a) For every* $\text{NRA}_1^{=}$*, there exists a* DRsA *accepting the same language.*

(b) For every $\text{URA}_1^{=}$*, there exists a* DRsA *accepting the same language.*

Proof. (a) Let $\mathcal{A}$ be an $\text{NRA}_1^{=}$ and $\mathcal{A}_r$ be its register-local version. Because $\mathcal{A}_r$ contains no disequality guards, the only way how Algorithm 1 could fail is at Line 15. Since $\mathcal{A}$ used at most one register r, then each state q of $\mathcal{A}_r$ will also use at most one register r_q (the

copy of r for q). Then P on Line 13 will only be a set of elements with no dependency. For such a set, the Cartesian abstraction is precise, so the test on Line 15 will never cause an abort.

(b) Let $\mathcal{A}$ be a $\text{URA}_1^=$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. First, using Fact 2.2.1, we construct an $\text{NRA}_1^=$ $\mathcal{A}_N$ accepting $\overline{\mathcal{L}}$ (note that although the first step in the construction in the proof of Fact 2.2.1 is to make $\mathcal{A}$ *complete*, we actually do not need to add transitions with missing guards (which might make us add transitions with a $\neq$ guard), but we can add transitions of the form $\cdot \xrightarrow{\cdot \mid \emptyset, \emptyset, \emptyset} q_{sink}$, which do not introduce $\neq$ guards). Then, we use (a) to convert $\mathcal{A}_N$ into a DRsA $\mathcal{A}_D$ accepting $\overline{\mathcal{L}}$. Finally, we complement $\mathcal{A}_D$ (by completing it and swapping final and non-final states), obtaining a DRsA accepting $\mathcal{L}$. □

Let $\mathcal{B}(\text{NRA}_1^=)$ be the class of languages that can be expressed using a Boolean combination of $\text{NRA}_1^=$ languages, i.e., it is the closure of $\text{NRA}_1^=$ languages under union, and intersection, and complement (it could also be denoted as $\mathcal{B}(\text{URA}_1^=)$).

Example 5.0.1. *For instance, the language* $L_{\exists,\neg\exists repeat}$*, defined as*

$$L_{\exists,\neg\exists repeat} = L_{\exists repeat} \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot L_{\neg\exists repeat}.$$

This language is composed as the concatenation of $L_{\exists repeat}$ *and* $L_{\neg\exists repeat}$ *with a delimiter, and it is in* $\mathcal{B}(\text{NRA}_1^=)$*, since it is the intersection of languages*

$$L_{\exists repeat} \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot \{\langle a, d\rangle \mid d \in \mathbb{D}\}^*$$

and

$$\{\langle a, d\rangle \mid d \in \mathbb{D}\}^* \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot L_{\neg\exists repeat},$$

but is expressible neither by an NRA *nor by a* URA *(*URA*s cannot express the part* before *the delimiter and* NRA*s cannot express the part* after *the delimiter).* □

From Theorem 5.0.2 we can conclude that $\mathcal{B}(\text{NRA}_1^=)$ is captured by DRsA.

Corollary 5.0.1. *For any language in* $\mathcal{B}(\text{NRA}_1^=)$*, there exists a* DRsA *accepting it.*

Proof. Let $L \in \mathcal{B}(\text{NRA}_1)$ and t be a term describing L using unions, intersections, and complements, with atoms being $\text{NRA}_1^=$ languages. W.l.o.g., we can assume that t is in the negation normal form (i.e., complements are only over atoms—it is easy to transform any term into this form using De Morgan's laws and double-complement elimination).

We can construct a DRsA $\mathcal{A}_D$ accepting L inductively as follows:

1. *Non-complemented literals*: if the literal is not complemented, we use Theorem 5.0.2(a) to obtain a DRsA accepting it.
2. *Complemented literals*: first, we use Fact 2.2.1 to obtain the $\text{URA}_1^=$ accepting the complemented language and then use Theorem 5.0.2(b) to convert it into a DRsA.
3. *Unions and intersections*: we simply use Theorem 3.2.3 to obtain the resulting DRsA.

□

Corollary 5.0.2. *The inclusion problem between* RsA *and* $\mathcal{B}(\text{NRA}_1^=)$ *is decidable.*

Proof. We just write $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ as $\mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)} = \emptyset$ and use Corollary 5.0.1 and Theorems 3.1.1 and 3.2.1. □

Chapter 6

Expressive Power

In this chapter, the model of register set automaton is positioned in the landscape of automata over data words, in terms of their expressive power. For a given formal model, its expressive power is the broadness of languages the model can recognize. This quality is yet another factor that can be used for comparing different automata models and creating various hierarchies. When analyzing the expressive power of automata models, they are usually classified with respect to the *Chomsky hierarchy* of languages [7]. In case of register set automata, we are looking to find out how expressive the model is in comparison to other register automata models with decidable emptiness problem. Since the Chomsky hierarchy is not used for classification of automata operating on data languages, we do not focus on finding the RsA's position in this hierarchy.

It is necessary to note that a greater expressive power often brings non-closure under some Boolean operations, or results in higher complexities of decision problems, which may be even undecidable for a given model. In this chapter, we first introduce the notions necessary for understanding the orderings between individual automata models, and then focus on showing the relationship between RsAs and other register automata models.

Let $\mathcal{L}_{\mathcal{A}}$ denote the set of all languages recognized by an automaton model $\mathcal{A}$, and $\mathcal{L}_{\mathcal{B}}$ denote the set of all languages recognized by an automaton model $\mathcal{B}$.

Definition 6.0.1. *A model $\mathcal{A}$ is said to be* more expressive *than the model $\mathcal{B}$ if it holds that $\mathcal{L}_{\mathcal{B}} \subseteq \mathcal{L}_{\mathcal{A}}$. Two automata are said to be* incomparable *in their expressive power if it holds that $\mathcal{L}_{\mathcal{A}} \nsubseteq \mathcal{L}_{\mathcal{B}}$, and, at the same time, $\mathcal{L}_{\mathcal{B}} \nsubseteq \mathcal{L}_{\mathcal{A}}$.*

6.1 Classifying the Expressive Power of RsA

In the following, we examine the abilities of individual register models to recognize chosen languages, thus comparing them with the model of register set automaton. We establish the orderings between given models by identifying specific languages expressible by one model and inexpressible by the other. For this, we use the languages introduced in Section 2.1.1.

Proposition 6.1.1. RsA *and* ARA_1 *are incomparable.*

Proof. Consider the language $L_{\forall a \exists b}$ from [10, Example 2.2]. Intuitively, $L_{\forall a \exists b}$ denotes the language of words where no two a-positions contain the same data value and every position with label a is followed by a matching b-position. As seen in Figure 6.1, $L_{\forall a \exists b}$ is recognizable by ARA_1 [10, Example 2.6], but is not recognizable by any URA, NRA, or RsA. Intuitively,

Table 6.1: Table showing the closure properties for a selection of register automata models as well as the decidability of some decision problems.

	RsA	RsA_n	DRsA	DRsA_n	NRA	DRA	URA	ARA_1	$\text{ARA}_1(\mathsf{g},\mathsf{s})$
union	✓	✓	✓	✗	✓	✓	✓	✓	✓
intersection	✓	✗	✓	✗	✓	✓	✓	✓	✓
complement	✗	✗	✓	✓	✗	✓	✗	✓	✗
universality	✗	✗	✓	✓	✗	✓	✓	✓	✗
emptiness	✓	✓	✓	✓	✓	✓	✗	✓	✓

if RsA was capable of recognizing this language, it would have to remember values seen with a in one register, and values seen with b in another register. The automaton would then have to be capable of matching the values in one register with the values in the other, thus making sure that no value seen with the label a was left unmatched. There is clearly no possibility of RsA representing this property. On the other hand, consider the language $L_{\neg\exists a\text{-}no\text{-}b}$ from [14, Proof of Proposition 3.2]. This language is not expressible by ARA_1. However, as seen in Figure 6.2b, even DRsA_1 with only one state is capable of recognizing this language. This shows that in addition to RsA and ARA_1 being incomparable, DRsA_1 and ARA_1 are incomparable too. □

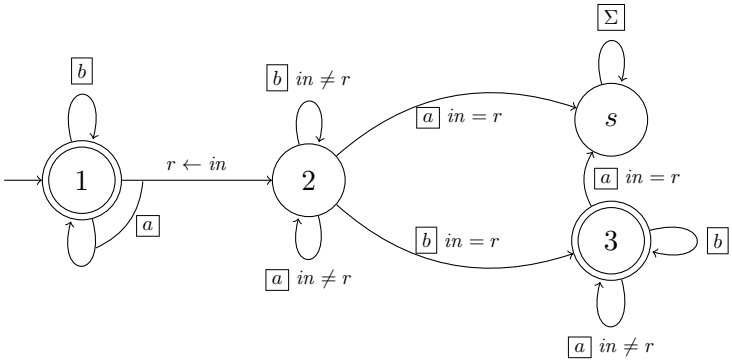


Figure 6.1: ARA_1, recognizing $L_{\forall a \exists b}$

Proposition 6.1.2. RsA *and* $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$ *are incomparable.*

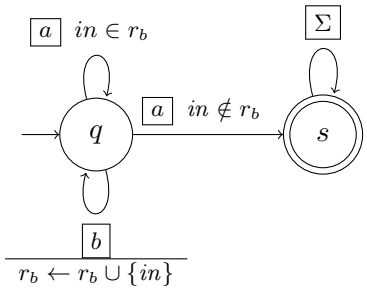
Proof. First, consider the language over $\Sigma = \{a, b\}$, $L_{\exists a\text{-}no\text{-}b}$ and its complement $L_{\neg\exists a\text{-}no\text{-}b}$. Intuitively, $L_{\exists a\text{-}no\text{-}b}$ is the language of words w such that there exists an input element $\langle a, d\rangle$ that is not preceded by an occurrence of a $\langle b, d\rangle$ element. Neither $L_{\exists a\text{-}no\text{-}b}$ nor $L_{\neg\exists a\text{-}no\text{-}b}$ can be accepted by ARA_1 while $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$ accepts $L_{\exists a\text{-}no\text{-}b}$. $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$ cannot accept the complement of this language, namely, $L_{\neg\exists a\text{-}no\text{-}b}$. On the other hand, as shown in Figures 6.2a and 6.2b, DRsA_1 can accept both $L_{\exists a\text{-}no\text{-}b}$ and $L_{\neg\exists a\text{-}no\text{-}b}$. As shown in Section 6.1, ARA_1 is capable of recognizing $L_{\forall a\exists b}$, while RsA is not. Since $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$ is more expressive than ARA_1 [14], the same holds for $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$. Therefore, RsA and $\text{ARA}_1(\mathsf{guess},\mathsf{spread})$ are incomparable. □

It might seem suspicious that DRsA_1 can express the language $L_{\neg\exists a\text{-}no\text{-}b}$. According to [14, Proof of Proposition 3.2], if an ARA_1 $\mathcal{A}$ could accept the language, $\mathcal{A}$ could be used

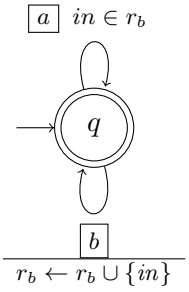
Table 6.2: Distinguishing languages for a selection of register automata models. Grey cells denote that the result is implied from the class being a sub/super-class of another class where the result is established. $\mathrm{DRA}_1^{(=)}$ denotes both DRA_1 and $\mathrm{DRA}_1^{=}$, similarly for $\mathrm{URA}_1^{(=)}$. $\mathrm{DRsA}_{(1)}$ denotes both DRsA and DRsA_1. None of the languages is accepted by DRA.

Language	$\mathrm{NRA}_1^{(=)}$	$\mathrm{URA}_1^{(=)}$	$\mathcal{B}(\mathrm{NRA}_1^{=})$	ARA_1	$\mathrm{DRsA}_{(1)}$	RsA_1	$\mathrm{ARA}_1(\mathsf{g},\mathsf{s})$
$L_{\exists repeat}$	✓ Ex. 2.2.2	✗ Ex. 2.2.2	✓	✓	✓ Ex. 3.0.1	✓	✓
$L_{\neg\exists repeat}$	✗ Ex. 2.2.2	✓ Ex. 2.2.2	✓	✓	✓ Ex. 3.0.4	✓	✓
$L_{\exists,\neg\exists repeat}$	✗ Ex. 5.0.1	✗ Ex. 5.0.1	✓ Ex. 5.0.1	✓	✓	✓	✓
$L_{\forall repeat}$	✗	✗	✗	✗ [14]	✗ Thm. 3.3.1	✗ Thm. 3.2.1	✗ [14]
$L_{\neg\forall repeat}$	✗	✗	✗	✗ [14]	✗ Thm. 3.3.1	✓ Ex. 3.0.2	✓ [14]
$L_{\exists a\text{-}no\text{-}b}$	✗	✗	✗	✗	✓ Ex. 6.1	✓	✓ [14]
$L_{\neg\exists a\text{-}no\text{-}b}$	✗	✗	✗	✗	✓ Ex. 6.1	✓	✗ [14]
$L_{\forall a\exists b}$	✗ Ex. 6.1	✗ Ex. 6.1	✗	✓ [10]	✗	✗ Ex. 6.1	✓

to decide language-emptiness of a Minsky machine (this problem is known to be undecidable even with an alphabet consisting of one symbol). So how come that we can express $L_{\neg\exists a\text{-}no\text{-}b}$ using DRsA_1, which have a decidable emptiness problem (cf. Theorem 3.1.1)? The reason is that in the construction of the automaton representing the accepting runs of a Minsky machine from [14], apart from $L_{\neg\exists a\text{-}no\text{-}b}$, we also need to be able to express the property "*every counter increment is matched with its decrement*", which is not expressible by RsA (cf. the proof of Theorem 3.2.1).



(a) A DRsA_1 recognizing $L_{\exists a\text{-}no\text{-}b}$



(b) A DRsA_1 recognizing $L_{\neg\exists a\text{-}no\text{-}b}$

Figure 6.2: Example DRsA_1

Remark 6.1.1. RsA*s are also incomparable to the class of* pebble automata *[29], since* DRsA*s generalize* DRA*s, as shown in [43, Remark 3.7], they can accept a language*

$$R^+ = \bigcup_{m=1,2,3\ldots} R_m^+,$$

where each R_m^+ *is defined as:*

$$c_0c_1\underbrace{\cdots}_{u_1}c_1c_2\underbrace{\cdots}_{u_2}c_2c_3\cdots\cdots\cdots c_{m-3}c_{m-2}\underbrace{\cdots}_{u_{m-2}}c_{m-2}c_{m-1}\underbrace{\cdots}_{u_{m-1}}c_{m-1}c_m,$$

where for each $i \in \{0,1,2\ldots m-1\}$ *the symbol* c_i *does not appear in* u_i *and* $c_i \neq c_{i+1}$. *This language is not expressible by* PA*s (cf. [43]). On the other hand,* RsA*s cannot express* $L_{\forall repeat}$*, which is expressible by* PA*s.* □

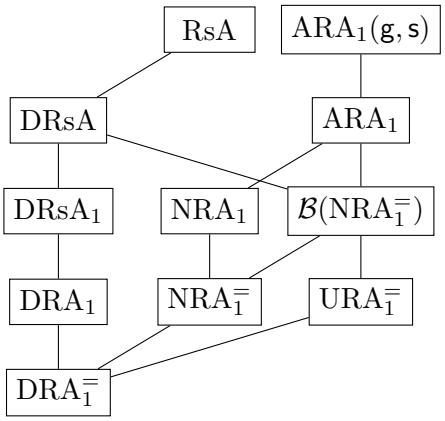


Figure 6.3: Hasse diagram comparing the expressive power of a selection of register automata models with decidable emptiness problem. All inclusions are strict. Languages distinguishing the different models can be found in Table 6.2.

The Hasse diagram comparing the expressive power of a selection of register automata models with decidable emptiness problem is in Figure 6.3 and languages that distinguish the various classes are in Table 6.2. Additionally, Table 6.1 contains the closure properties and decidability of some decision problems for selected register models. It can be seen that the RsAs are incomparable in expressive power to other popular automata models over data words, such as alternating register automata and their extension $\mathrm{ARA}(\mathsf{guess},\mathsf{spread})$.

Chapter 7

Extensions of Register Set Automata

In this chapter, we present several extensions of the register set automata model. These models are based on extending the supported operations in manipulation with registers on the transitions of the automaton. It can be seen that even a slight modification results in undecidability of some of the decisions problems. The following sections contain the formal definitions of each extension and the main results for each of these models.

7.1 RsAs with Register Emptiness Test

Definition 7.1.1. *A* register set automaton with register emptiness test *(*RsA$^{=\emptyset}$*) is a tuple* $\mathcal{A}_E = (Q, \mathbf{R}, \Delta^{=\emptyset}, I, F)$ *where* $Q, \mathbf{R}, I, F$ *are the same as for* RsA*s and the transition relation* $\Delta^{=\emptyset}$ *for* RsA$^{=\emptyset}$ *is defined as* $\Delta^{=\emptyset} \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$.

The **semantics of a transition** $q \xrightarrow{a \mid g^{\in}, g^{\notin}, g^{=\emptyset}, up} s$ is such that:

- $\mathcal{A}_E$ can move from state q to state s if
  - the Σ-symbol at the current position of the input word is a,
  - the $\mathbb{D}$-symbol at the current position is in all registers from $g^{\in}$,
  - the $\mathbb{D}$-symbol at the current position is in no register from $g^{\notin}$, and
  - all registers from $g^{=\emptyset}$ are empty.
- The content of the registers is updated so that $r_i \leftarrow \bigcup\{x \mid x \in up(r_i)\}$.

Lemma 7.1.1. *For every* RsA$^{=\emptyset}$ $\mathcal{A}$*, there exists an* RsA $\mathcal{A}'$ *with the same language.*

Proof. Proof is done by showing the construction of an RsA $\mathcal{A}'$.

The modification of $\mathcal{A}'$ appears in the structure of states, where we code the information about the empty registers into the states themselves, and modify the transition relation accordingly. Let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$. The information about the emptiness of registers is kept in the form of a binary vector, denoted $\mathbf{v}_\emptyset$, such that $\mathbf{v}_\emptyset[r_i] = 0$ iff $r_i = \emptyset$, and $\mathbf{v}_\emptyset[r_i] = 1$ otherwise. Let $\mathbf{V}_\emptyset = \{(e_1, \dots e_n) \mid \forall 1 \leq i \leq n : e_i \in \{0, 1\} \wedge n = |\mathbf{R}|\}$ denote the set of all possible binary vectors. The RsA equivalent to $\mathcal{A}$ is created as $\mathcal{A}' = (Q', \mathbf{R}, \Delta', I', F')$ where $Q' = Q \times \mathbf{V}_\emptyset$, $I' = \{(q_i, \mathbf{v}^0_\emptyset) \mid q_i \in I \wedge \forall r_i \in \mathbf{R} : \mathbf{v}_\emptyset[r_i] = 0)\}$, $F' = \{(q_f, \mathbf{v}_\emptyset) \mid q_f \in F\}$, and $\Delta' = \{(q_1, \mathbf{v}^1_\emptyset) \xrightarrow{a \mid g^{\in}, g^{\notin}, up} (q_2, \mathbf{v}^2_\emptyset) \mid q_1 \xrightarrow{a \mid g^{\in}, g^{\notin}, S, up} q_2 \in \Delta \wedge S = \{r \in \mathbf{R} \mid$

$\mathbf{v}^1_\emptyset = 0\} \wedge \forall r_i \in \mathbf{R} : \mathbf{v}^2_\emptyset[r_i] = up_\emptyset(\mathbf{v}^1_\emptyset[r_i])\}$. For all $1 \leq i \leq |\mathbf{R}|$ and given register r, the update of vector $\mathbf{v}^i_\emptyset[r]$, denoted by $up_\emptyset(\mathbf{v}^i_\emptyset[r])$ is defined as:

$$up_\emptyset(\mathbf{v}^i_\emptyset[r]) = \begin{cases} 1 & \text{if } (\exists r_j \in \mathbf{R} : r_j \in up(r) \wedge \mathbf{v}^{i-1}_\emptyset[r_j] \neq 0) \vee \\ & (in \in up(r)), \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{7.1}$$

For such constructed RsA $\mathcal{A}'$ it holds that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$. □

7.2 RsAs with Register Equality Test

Definition 7.2.1. *A* register set automaton with register equality test *(*RsA$^{=r}$*) is a tuple* $\mathcal{A}_{Eq} = (Q, \mathbf{R}, \Delta^{=r}, I, F)$ *where* $Q, \mathbf{R}, I, F$ *are the same as for* RsA *and the transition relation* $\Delta^{=r}$ *for* RsA$^{=r}$ *is defined as* $\Delta^{=r} \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R}}) \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$.

The **semantics of a transition** $q \xrightarrow{a \mid g^{\in}, g^{\notin}, g^{=}, up} s$ is such that:

- $\mathcal{A}_E$ can move from state q to state s if
  - the Σ-symbol at the current position of the input word is a,
  - the $\mathbb{D}$-symbol at the current position is in all registers from $g^{\in}$,
  - the symbol at the current position is in no register from $g^{\notin}$, and
  - for all $r \in \mathbf{R}$ it holds that if $r_i \in g^{=}(r)$, then $r_i = r$.
- The content of the registers is updated so that $r_i \leftarrow \bigcup\{x \mid x \in up(r_i)\}$.

Theorem 7.2.1. *The emptiness problem for* RsA$^{=r}$ *is undecidable.*

Proof. The proof is done by reduction from *reachability* in *Petri nets* with *inhibitor arcs* (PN$_I$), which is an undecidable problem [31]. Given a PN$_I$ $\mathcal{N}_I$ with inhibitor arcs, we construct a corresponding RsA$^{=r}$ $\mathcal{A}^{=r}_I$. The process of construction of $\mathcal{A}^{=r}_I$ follows the reduction of TPN to RsA in the proof of Lemma 4.3.2. The structure of the resulting automaton differs in *protogadgets* whose concatenation is used for construction of *gadgets* that make up the reduced RsA$^{=r}$.

The following *protogadgets* are used in the reduction:

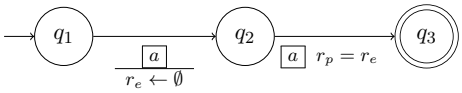
1. The *EmptyEq* protogadget (depicted in Figure 7.1a), which simulates the inhibitor arc leading from place p is an RsA$^{=r}$ defined as:

   $$\textsc{EmptyEq}(p) = (\{q_1, q_2, q_3\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \emptyset, \emptyset, \{r_e \mapsto \emptyset\}} q_2, q_2 \xrightarrow{a \mid \emptyset, \emptyset, \{r_p \mapsto \{r_e\}\}, \emptyset} q_3\}, \{q_1\}, \{q_3\}).$$

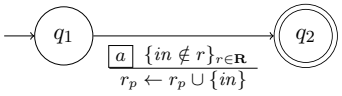
   Intuitively, EmptyEq simulates a register emptiness test. At first, the protogadget explicitly assigns the value of the empty set to the register r_e and then it compares its equality with the content of the register representing the place in which the inhibitor arc originates.

2. The *New Token* protogadget (depicted in Figure 7.1b), which simulates adding a token to a place p, is an RsA$^{rm}_{=\emptyset}$ defined in the following way:

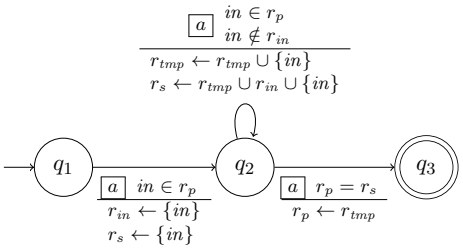
   $$\textsc{NewToken}(p) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \mathbf{R}, \emptyset, \{r_p \mapsto \{r_p, in\}\}} q_2\}, \{q_1\}, \{q_2\}).$$



(a) The EmptyEq(p) protogadget.



(b) The NewToken(p) protogadget.



(c) The NonLossyRm(p) protogadget.

Figure 7.1: Protogadgets used in the construction of the RsA$^{=r}$ for $\mathcal{N}_I$.

Intuitively, for each arc originating in the transition and ending in a particular place, a token is added to the register representing the destination. The guard ensures that the added value is not already present within the register, so that the number of values actually increases.

3. The *Non-lossy Remove Token* protogadget (depicted in Figure 7.1c), which simulates removal of a token from a place p is an RsA$^{=r}$ defined in the following way:

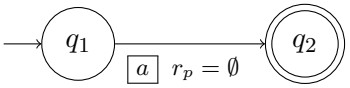
$$\begin{aligned}\textsc{NonLossyRm}(p) = (\{q_1, q_2, q_3\}, \mathbf{R}, \{&q_1 \xrightarrow{a \mid \{r_p\}, \emptyset, \emptyset, \{r_{in} \mapsto \{in\}, r_s \mapsto \{in\}\}} q_2,\\ &q_2 \xrightarrow{a \mid \{r_p\}, \{r_{in}\}, \emptyset, \{r_{tmp} \mapsto \{r_{tmp}, inp\}, r_s \mapsto \{r_{tmp}, r_{in}, in\}\}} q_2,\\ &q_2 \xrightarrow{a \mid \emptyset, \emptyset, \{r_p \mapsto \{r_s\}\}, \{r_p \mapsto \{r_{tmp}\}\}} q_3\}, \{q_1\}, \{q_2\}).\end{aligned}$$

Intuitively, for each arc originating in a place p and terminating in a PN$_I$ transition, the respective number of values has to be removed from the register representing place p. Therefore, on each *protogadget* of this kind, one value is removed from a register representing the source place in a lossless manner. The quality of being lossless is necessary in order to sustain the semantics of the source PN$_I$. Otherwise, some transitions may be enabled even though they were not enabled in the source PN$_I$. □

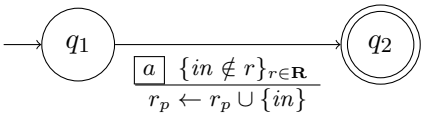
To conclude, it can be observed that even small extension of RsA which allows for testing the equality of registers on the transitions of the automaton leads to undecidability of the emptiness problem.

7.3 RsAs with Removal and Register Emptiness Test

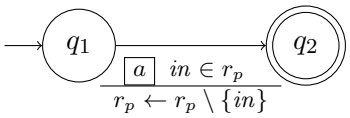
Definition 7.3.1. *An* RsA *with* removal and register emptiness test *(*RsA$^{rm}_{=\emptyset}$*) is a tuple* $\mathcal{A}_{RE} = (Q, \mathbf{R}, \Delta^{rm}_{=\emptyset}, I, F)$*, where* $Q, \mathbf{R}, I, F$ *are the same as for* RsA*s and the transition relation* $\Delta^{rm}_{=\emptyset}$ *is defined as* $\Delta^{rm}_{=\emptyset} \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$.



(a) The Empty(p) protogadget.



(b) The NewToken(p) protogadget.



(c) The TokenRem(p) protogadget.

Figure 7.2: Protogadgets used in the construction of the $\text{RsA}^{rm}_{=\emptyset}$ for $\mathcal{N}_I$.

The **semantics of a transition** $q \xrightarrow{a \mid g^{\in}, g^{\notin}, g^{=\emptyset}, rem, up} s$ is such that:

- $\mathcal{A}_{RE}$ can move from state q to state s if
  - the Σ-symbol at the current position of the input word is a,
  - the $\mathbb{D}$-symbol at the current position is in all registers from $g^{\in}$,
  - he $\mathbb{D}$-symbol at the current position is in no register from $g^{\notin}$, and
  - all registers from $g^{=\emptyset}$ are empty.
- Regarding *rem* and *up*, the content of the registers is updated so that for all r_i from *rem*, first, $r_i \leftarrow \bigcup\{x \mid x \in up(r_i)\}$ and then $r_i \leftarrow r_i \setminus \{in\}$.

Theorem 7.3.1. *The emptiness problem for* $\text{RsA}^{rm}_{=\emptyset}$ *is undecidable.*

Proof. The proof is done by showing the reducibility from *reachability* in *Petri nets* with *inhibitor arcs*, which is an undecidable problem.

Given a PN_I $\mathcal{N}_I$ with inhibitor arc, we construct the $\text{RsA}^{rm}_{=\emptyset}$ $\mathcal{A}_{\mathcal{N}_I}$. The structure of $\mathcal{A}_{\mathcal{N}_I}$ is similar to the structure of RsA $\mathcal{A}_{\mathcal{N}}$ in the proof of Theorem 7.2.1.

The only difference is in *gadgets* used for simulation of respective transition in PN_I and gadget for doing the reachability test. These are created by concatenation of respective *protogadgets*, which are defined in the following way:

1. The *Empty* protogadget (depicted in Figure 7.2a), which simulates the inhibitor arc leading from the place p is an $\text{RsA}^{rm}_{=\emptyset}$ defined as:

$$\textsc{Empty}(p) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \emptyset, \{r_p\}, \emptyset, \{r_p \mapsto \emptyset\}} q_2\}, \{q_1\}, \{q_2\}).$$

   Intuitively, since the inhibitor arc enables its transition only if the source place is empty, the respective *protogadget* checks on its guard whether the register representing the source place is empty as well.

2. The *New Token* protogadget (depicted in Figure 7.2b), which simulates adding a token to a place p, is an $\text{RsA}^{rm}_{=\emptyset}$ defined in the following way:

$$\textsc{NewToken}(p) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \mathbf{R}, \emptyset, \emptyset, \{r_p \mapsto \{r_p, in\}\}} q_2\}, \{q_1\}, \{q_2\}).$$

Intuitively, for each arc originating in the transition and ending in a particular place, a token is added to the register representing the destination. The guard ensures that the added value is not already present within the register, so that the number of values actually increases.

3. The *Remove Token* protogadget (depicted in Figure 7.2c), which simulates the removal of a token from a place p, is an $\mathrm{RsA}^{rm}_{=\emptyset}$ defined in the following way:

$$\textsc{TokenRem}(p) = (\{q_1, q_2\}, \mathbf{R}, \{q_1 \xrightarrow{a \mid \emptyset, \{r_p\}, \emptyset, \{r_p\}, \{r_p \mapsto \{r_p\}\}} q_2\}, \{q_1\}, \{q_2\}).$$

Intuitively, for each token removed from the source place, one value is removed from the register representing that place. □

To conclude, it can be seen that extending the model of RsA with possibility of removing values from registers and testing registers for emptiness brings the undecidability of the emptiness problem.

Chapter 8

Conclusion

In this thesis, we have introduced a model of register set automaton, based on extending the model of register automaton. We allow for storing of a set of values in each register, contrary to the single value allowed by RA. In addition to the formal definition of the introduced model, we have proven its closure properties, with restriction to both fixed number of registers, and a deterministic structure. We have shown the decidability of the emptiness testing of RsA, and proven its $\mathbf{F}_\omega$-completeness by interreducibility with coverability testing of transfer Petri nets.

We have provided a semi-algorithm for transforming a subclass of register automata into deterministic register set automata. Furthermore, a comparison of the register set automata and other register models was explored in context of their expressive power. We have shown the incomparability of RsAs and alternating register automata, together with their extensions.

Finally, we observed some of the possible extensions of register set automata and decidability of their emptiness problem. This was proven undecidable for RsAs allowing for removal of values from registers, and testing the register emptiness on the transitions, as well as for the register equality testing. The undecidability was proven by reduction from reachability in Petri nets with inhibitor arcs. Additionally, we have shown that for each RsA with extension to register emptiness test, there exists an equivalent RsA accepting the same language.

There are many challenges we wish to address in the future. We would like to focus on exploring different structures of the registers, which would allows us, in case of pattern matching with backreferences, to work with dependencies between capture groups, and with capture groups that can hold more than one symbol. Additionally, we would like to propose algorithms for efficient testing on language inclusion of register automata, improve our determinisation algorithm to work on a larger class of input NRAs, and identify a logic fragment corresponding to (D)RsA.

Bibliography

[1] Aaronson, S., Kuperberg, G. and Granade, C. *The complexity zoo* [https://complexityzoo.net/Complexity_Zoo]. 2005. [Online; accessed 10-May-2022].

[2] Adar Weidman. *Regular expression Denial of Service — ReDoS* [https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS]. 2021. [Online; accessed 19-December-2021].

[3] Arora, S. and Barak, B. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] Becchi, M. and Crowley, P. Extending finite automata to efficiently match Perl-compatible regular expressions. In: *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*. Madrid, Spain: ACM Press, 2008, p. 1–12. DOI: 10.1145/1544012.1544037. ISBN 978-1-60558-210-8. Available at: http://portal.acm.org/citation.cfm?doid=1544012.1544037.

[5] Bollig, B., Habermehl, P., Leucker, M. and Monmege, B. A Fresh Approach to Learning Register Automata. In: *Developments in Language Theory*. Berlin, Heidelberg: Springer, 2013, p. 118–130. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-38771-5_12. ISBN 978-3-642-38771-5.

[6] Chen, Y., Havlena, V., Lengál, O. and Turrini, A. A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving. In: *Programming Languages and Systems*. Cham: Springer International Publishing, 2020, vol. 12470, p. 343–363. DOI: 10.1007/978-3-030-64437-6_18. ISBN 978-3-030-64436-9 978-3-030-64437-6. Series Title: Lecture Notes in Computer Science. Available at: https://link.springer.com/10.1007/978-3-030-64437-6_18.

[7] Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*. 1956, vol. 2, no. 3, p. 113–124. DOI: 10.1109/TIT.1956.1056813.

[8] Clemente, L., Lasota, S. and Piórkowski, R. Determinisability of register and timed automata. *ArXiv:2104.03690 [cs]*. april 2021. Available at: http://arxiv.org/abs/2104.03690.

[9] D'Antoni, L. and Veanes, M. Minimization of symbolic automata. In: Jagannathan, S. and Sewell, P., ed. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014, p. 541–554. DOI: 10.1145/2535838.2535849. Available at: https://doi.org/10.1145/2535838.2535849.

[10] Demri, S. and Lazić, R. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*. april 2009, vol. 10, no. 3, p. 1–30. DOI: 10.1145/1507244.1507246. ISSN 1529-3785, 1557-945X. Available at: https://dl.acm.org/doi/10.1145/1507244.1507246.

[11] Emerson, E. A. and Namjoshi, K. S. On model checking for non-deterministic infinite-state systems. In: IEEE. *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 98CB36226)*. 1998, p. 70–80.

[12] Esparza, J., Finkel, A. and Mayr, R. On the Verification of Broadcast Protocols. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 1999, p. 352–359. DOI: 10.1109/LICS.1999.782630. Available at: https://doi.org/10.1109/LICS.1999.782630.

[13] Fairtlough, M. and Wainer, S. Ordinal complexity of recursive definitions. *Information and Computation*. 1992, vol. 99, no. 2, p. 123–153. DOI: https://doi.org/10.1016/0890-5401(92)90027-D. ISSN 0890-5401. Available at: https://www.sciencedirect.com/science/article/pii/089054019290027D.

[14] Figueira, D. Alternating register automata on finite words and trees. *Logical Methods in Computer Science*. march 2012, vol. 8, no. 1, p. 22. DOI: 10.2168/LMCS-8(1:22)2012. ISSN 18605974. Available at: http://arxiv.org/abs/1202.3957.

[15] Grigore, R., Distefano, D., Petersen, R. L. and Tzevelekos, N. Runtime Verification Based on Register Automata. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer, 2013, p. 260–276. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-36742-7_19. ISBN 978-3-642-36742-7.

[16] Hofman, P. and Totzke, P. Trace inclusion for one-counter nets revisited. In: Springer. *International Workshop on Reachability Problems*. 2014, p. 151–162.

[17] Holík, L., Lengál, O., Saarikivi, O., Turoňová, L., Veanes, M. and Vojnar, T. Succinct Determinisation of Counting Automata via Sphere Construction. In: Lin, A. W., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2019, p. 468–489. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-34175-6_24. ISBN 978-3-030-34175-6.

[18] Hopcroft, J. E., Motwani, R. and Ullman, J. D. Introduction to automata theory, languages, and computation. *ACM Sigact News*. ACM New York, NY, USA. 2001, vol. 32, no. 1, p. 60–65.

[19] Hunter Monroe. *Complexity Garden* [https://complexityzoo.net/Complexity_Garden]. 2005. [Online; accessed 10-May-2022].

[20] Jančar, P. Nonprimitive recursive complexity and undecidability for Petri net equivalences. *Theoretical Computer Science*. 2001, vol. 256, no. 1, p. 23–30. DOI: https://doi.org/10.1016/S0304-3975(00)00100-6. ISSN 0304-3975. ISS. Available at: https://www.sciencedirect.com/science/article/pii/S0304397500001006.

[21] JOHNSON, D. S. A Catalog of Complexity Classes. In: VAN LEEUWEN, J., ed. *Algorithms and Complexity*. Amsterdam: Elsevier, 1990, p. 67–161. Handbook of Theoretical Computer Science. DOI: https://doi.org/10.1016/B978-0-444-88071-0.50007-2. ISBN 978-0-444-88071-0. Available at: https://www.sciencedirect.com/science/article/pii/B9780444880710500072.

[22] KAMINSKI, M. and FRANCEZ, N. Finite-memory automata. *Theoretical Computer Science*. november 1994, vol. 134, no. 2, p. 329–363. DOI: 10.1016/0304-3975(94)90242-9. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/0304397594902429.

[23] KLEENE, S. C. Representation of events in nerve nets and finite automata. *Automata studies*. Princeton, NJ. 1956, vol. 34, p. 3–41.

[24] KLIN, B., LASOTA, S. and TORUNCZYK, S. Nondeterministic and co-Nondeterministic Implies Deterministic, for Data Languages. In: *FoSSaCS*. 2021, p. 365–384.

[25] LÖB, M. H. and WAINER, S. S. Hierarchies of number-theoretic functions. I. *Archiv für mathematische Logik und Grundlagenforschung*. Springer. 1970, vol. 13, no. 1, p. 39–51.

[26] MANUEL, A. and RAMANUJAM, R. Automata over infinite alphabets. In: *Modern applications of automata theory*. World Scientific, 2012, p. 529–553.

[27] MEYER, A. R. and RITCHIE, D. M. The Complexity of Loop Programs. In: *Proceedings of the 1967 22nd National Conference*. New York, NY, USA: Association for Computing Machinery, 1967, p. 465–469. ACM '67. DOI: 10.1145/800196.806014. ISBN 9781450374941. Available at: https://doi.org/10.1145/800196.806014.

[28] MINSKY, M. L. Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines. *Annals of Mathematics*. 1961, vol. 74, no. 3, p. 437–455. DOI: 10.2307/1970290. ISSN 0003-486X. Available at: https://www.jstor.org/stable/1970290.

[29] NEVEN, F., SCHWENTICK, T. and VIANU, V. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 2004, vol. 5, no. 3, p. 403–435. DOI: 10.1145/1013560.1013562. Available at: https://doi.org/10.1145/1013560.1013562.

[30] RABIN, M. O. and SCOTT, D. S. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 1959, vol. 3, no. 2, p. 114–125. DOI: 10.1147/rd.32.0114. Available at: https://doi.org/10.1147/rd.32.0114.

[31] REINHARDT, K. Reachability in Petri nets with inhibitor arcs. *Electronic Notes in Theoretical Computer Science*. Elsevier. 2008, vol. 223, p. 239–264.

[32] ROESCH, M. et al. *Snort: A Network Intrusion Detection and Prevention System* [http://www.snort.org]. 2022. [Online; accessed 06-January-2022].

[33] SAKAMOTO, H. and IKEDA, D. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 2000, vol. 231, no. 2, p. 297–308. DOI:

10.1016/S0304-3975(99)00105-X. Available at: https://doi.org/10.1016/S0304-3975(99)00105-X.

[34] Schmid, M. L. Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing. august 2016, vol. 249, p. 1–17. DOI: 10.1016/j.ic.2016.02.003. Available at: https://www.sciencedirect.com/science/article/pii/S0890540116000109.

[35] Schmitz, S. Complexity Hierarchies Beyond Elementary. *ACM Transactions on Computation Theory*. february 2016, vol. 8, no. 1, p. 1–36. DOI: 10.1145/2858784. ISSN 1942-3454, 1942-3462. Available at: http://arxiv.org/abs/1312.5686.

[36] Schmitz, S. *Algorithmic Complexity of Well-Quasi-Orders*. 2017. Habilitation à diriger des recherches. École normale supérieure Paris-Saclay. Available at: https://tel.archives-ouvertes.fr/tel-01663266.

[37] Schmitz, S. and Schnoebelen, P. *Algorithmic Aspects of WQO Theory*. August 2012. Available at: https://cel.archives-ouvertes.fr/cel-00727025.

[38] Schmitz, S. and Schnoebelen, P. The Power of Well-Structured Systems. In: D'Argenio, P. R. and Melgratti, H. C., ed. *CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Springer, 2013, vol. 8052, p. 5–24. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-40184-8_2. Available at: https://doi.org/10.1007/978-3-642-40184-8_2.

[39] Schnoebelen, P. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*. 2002, vol. 83, no. 5, p. 251–261. DOI: https://doi.org/10.1016/S0020-0190(01)00337-4. ISSN 0020-0190. Available at: https://www.sciencedirect.com/science/article/pii/S0020019001003374.

[40] Schnoebelen, P. Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In: Hlinený, P. and Kucera, A., ed. *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*. Springer, 2010, vol. 6281, p. 616–628. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-15155-2_54. Available at: https://doi.org/10.1007/978-3-642-15155-2_54.

[41] Sipser, M. Introduction to the Theory of Computation. *ACM Sigact News*. ACM New York, NY, USA. 1996, vol. 27, no. 1, p. 27–29.

[42] Stack Exchange. *Outage Postmortem* [http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016]. 2016. [Online; accessed 17-April-2022].

[43] Tan, T. Graph Reachability and Pebble Automata over Infinite Alphabets. *ACM Trans. Comput. Log.* 2013, vol. 14, no. 3, p. 19:1–19:31. DOI: 10.1145/2499937.2499940. Available at: https://doi.org/10.1145/2499937.2499940.

[44] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. and Vojnar, T. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages*. november 2020, vol. 4, OOPSLA, p. 218:1–218:30. DOI: 10.1145/3428286. Available at: https://doi.org/10.1145/3428286.

[45] Urquhart, A. The complexity of decision procedures in relevance logic II. *Journal of Symbolic Logic*. Cambridge University Press. 1999, vol. 64, no. 4, p. 1774–1802. DOI: 10.2307/2586811.

Appendix A

Content of the Attached Storage Medium

- `src-thesis/` Folder with LaTeX source files.
- `src-thesis/figures` Folder with figures used throughout the thesis.
- `gulcikova-rsa-thesis.pdf` Electronically submitted version of the thesis.
- `gulcikova-rsa-thesis-p.pdf` Printed version of the thesis[1].

[1] Content is the same as in the version for the electronic submission, difference is in the structure of pages, which was modified for printing on both sides of the paper.