



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ADVANCED APPROACHES FOR DATA LANGUAGES

POKROČILÉ METODY PRÁCE S DATOVÝMI JAZYKY

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. JAN VAŠÁK

SUPERVISOR

VEDOUCÍ PRÁCE

doc. Ing. ONDŘEJ LENGÁL, Ph.D.

BRNO 2026

Master's Thesis Assignment



172680

Institut: Department of Intelligent Systems (DITS)
Student: **Vašák Jan, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Mathematical Methods
Title: **Advanced Approaches for Data Languages**
Category: Theoretical Computer Science
Academic year: 2025/26

Assignment:

1. Study the theory of formal models for data languages, such as variants of register automata or logics. Focus on models that are suitable for an implementation of matching regular expressions with back-references or modeling programs with lists and sets.
2. Based on a discussion with the supervisor, choose at least one of the following options:
 - a) Establish and prove the theoretical properties of the considered models, e.g. decidability and complexity of problems such as emptiness, membership, language inclusion, or functional equivalence.
 - b) Carry out the following:
 - (i) Develop efficient algorithms for working with a chosen model, focusing on efficient handling of operations such as membership, inclusion testing, reduction, etc.
 - (ii) Implement the developed algorithms and compare them experimentally with algorithms that work on finite automata (for instance over an alphabet such as ASCII or Unicode).
3. Discuss the obtained results and possible further extensions.

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)
- Gulčíková, S. and Lengál, O., 2022. Register Set Automata (Technical Report). *arXiv preprint arXiv:2205.12114*.
- N. Tzevelekos and R. Grigore, "History-Register Automata," in *Foundations of Software Science and Computation Structures*, F. Pfenning, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 17–33. doi: [10.1007/978-3-642-37075-5_2](https://doi.org/10.1007/978-3-642-37075-5_2).

Requirements for the semestral defence:

Item 1.

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

Supervisor: **Lengál Ondřej, doc. Ing., Ph.D.**
Head of Department: Kočí Radek, Ing., Ph.D.
Beginning of work: 1.11.2025
Submission deadline: 27.5.2026
Approval date: 3.11.2025

Abstract

Matching regular expressions with back-references (regexes) is a common task in computer programs. To match these regexes, current state-of-the-art regex matchers use backtracking algorithms with exponential worst-case complexity, or do not support them at all. The exponential worst-case time complexity makes the backtracking matchers using backtracking vulnerable to the so-called regular expression denial of service attack (ReDoS), where an attacker submits a malicious text to a regex matcher running on a server, triggering a large slowdown and causing the server to become unresponsive. We present a robust regex matcher capable of matching a class of regexes with back-references in linear time (for a fixed number of capture groups). Our matcher is based on recently introduced model of register set automata, which are capable of representing the above-mentioned class of regexes with back-references deterministically. We tested the performance of our matcher under simulated ReDoS attacks on regexes from real-world projects and compared it against state-of-the-art tools. The experiments show that our matcher is immune to all tested ReDoS attacks; in contrast with the state-of-the-art matchers, which on these attacks suffer from extreme slowdowns.

Abstrakt

Regex matching se zpětnými odkazy je častá úloha v počítačových programech. Současné matchery ovšem regexy se zpětnými odkazy řeší backtracking algoritmy s exponenciální časovou složitostí v nejhorším případě, nebo je vůbec nepodporují. Kvůli této složitosti jsou matchery používající backtracking zranitelné tak zvanými ReDoS (regular expression denial of service) útoky. ReDoS útok nastává v případě, kdy útočník zadá škodlivý vstup regex matcheru běžícím na serveru a může tak díky zpomalení způsobit výpadek serveru. V této práci představíme regex matcher podporující třídu regexů se zpětnými odkazy s lineární časovou složitostí (pro pevný počet capture group). Naš matcher je založen na nedávno představených registr-množinových automatech, které umožňují deterministickou reprezentaci výše zmíněné třídy regexů se zpětnými odkazy. Výkon našeho matcheru jsme ověřili na simulovaných ReDoS útocích na regexy z praxe a porovnali výsledky s výkonem průmyslových matcherů. Výsledky experimentů prokázaly, že náš matcher je imunní vůči otestovaným ReDoS útokům. Oproti tomu, všem ostatním matcherům tyto ReDoS útoky způsobily extrémní zpomalení.

Keywords

regex matching, back-references, ReDoS, data words, register set automata

Klíčová slova

regex matching, zpětné odkazy, ReDoS, datová slova, register-množinové automaty

Reference

VÁŠÁK, Jan. *Advanced Approaches for Data Languages*. Brno, 2026. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Zpětné odkazy jsou často používaným rozšířením regulárních výrazů (regexů, z anglického **regular expression**). Problém matchingu regexů se zpětnými odkazy, tedy problém zjištění zda vstupní text odpovídá regexu, je ovšem výpočetně složitý (konkrétně NP-úplný). Proto současné matchery používají při výskytu zpětných odkazů backtracking algoritmy. Backtracking algoritmus spočívá ve vytvoření nedeterministického automatu reprezentující regex a následné simulace tohoto automatu pro daný vstup. Ovšem nedeterminismus způsobí, že je pro daný vstup nutné najít potenciální akceptující cestu v automatu. Algoritmus zkouší cesty dokud jsou možné a v případě selhání se navrácí do předchozího bodu aby vyzkoušel jinou možnost pokračování cesty — odsud název backtracking. Cest v automatu ovšem může být v nejhorším případě exponenciálně mnoho vůči délce vstupu, což znamená, že backtracking algoritmy mají v nejhorším případě exponenciální časovou složitost.

Tato složitost může způsobit problém např. pro regex matcher zpracovávající uživatelské vstupy na serveru. Pokud takovému regex matcheru uživatel záměrně dá škodlivý vstup, mluvíme o tak zvaném ReDoS (regular expression denial of service) útoku. ReDoS útoky jsou reálnou hrozbou, např. výpadek stránky StackOverflow v roce 2016 byl způsoben ReDoS útokem. Jelikož všechny současné matchery podporující zpětné odkazy pro ně používají backtracking, jsou nutně zranitelné vůči ReDoS útokům. Je vhodné zmínit, že matchery které se zaměřují na výkon a bezpečnost, např. RE2 a HyperScan, z tohoto důvodu zpětné odkazy nepodporují.

Registr-množinové automaty (RSA — register set automata) jsou automatový model umožňující deterministickou reprezentaci třídy regexů se zpětnými odkazy. RSA rozšiřují konečné automaty o konečný počet registrů, kde každý registr obsahuje množinu symbolů. Navíc v této práci připouštíme množiny symbolů na hranách RSA místo jednotlivých symbolů, což usnadňuje reprezentaci regexů. Reprezentace regexů funguje tak, že je regex převeden na odpovídající registrový automat (RA). Ovšem registrové automaty nelze obecně determinizovat, tedy RSA jsou použity pro jejich determinizaci (což umožní determinizaci výrazně větší třídy RA než při použití pouze RA). Deterministický RSA reprezentující regex lze poté odsimulovat nad vstupem pro provedení regex matchingu. Tato simulace je již deterministická, tedy existuje pouze jedna cesta v RSA pro daný vstup.

Tato práce představí nový regex matcher podporující třídu regexů se zpětnými odkazy, který je výrazným zlepšením oproti dříve existujícímu prototypu. Matcher pracuje formou virtuálního stroje (VM — virtual machine), tedy interpretuje kód reprezentující RSA a jako vstup dostává vstupní řetězec daného regex matching problému. Skládá se tedy ze dvou částí: samotný VM matcher a generátor kódu pro RSA reprezentující regexy se zpětnými odkazy.

Výše zmíněný prototyp matcheru založeného na RSA byl v rámci této práce významně zlepšen a výsledkem je výrazně efektivnější a robustní regex matcher. Prototyp byl implementován v jazyce Python a pracuje již naznačeným způsobem: regex —[konstrukce RA]→ nedeterministický RA —[determinizace]→ deterministický RSA —[simulace RSA]→ výsledek ano/ne. Optimalizace v této práci byly zaměřeny hlavně na konstrukci RA a determinizaci. Při konstrukci RA byl zredukován počet stavů a registrů konstruovaných RA. Determinizace byla optimalizována technikami specifickými pro RA vznikajících z regexů a jako výsledek byl snížen počet stavů a hran výsledného RSA.

V této práci navržený generátor kódu pracuje nad vylepšeným prototypem a spouští se po determinizaci. Tedy pro daný RSA je vygenerován kód který jej reprezentuje. Pro výběr přechodu podle množin symbolů definovaných na hranách RSA se používají prefixové stromy s listy označujícími množiny symbolů. Tyto prefixové stromy jsou definovány mimo

samotný kód a jsou předány VM matcheru do paměti. Pro jednu množinu symbolů ovšem může existovat více hran rozlišených testy registrů. Pro výběr hrany podle obsahu registrů byly naimplementovány multi-terminální rozhodovací diagramy (MTBDD — multi-terminal binary decision diagram). MTBDD používají jako terminální uzly konkrétní přechody RSA a v neterminálních uzlech se rozhoduje následník podle příslušnosti vstupního symbolu do registru spojeného s daným uzlem. Tedy tyto MTBDD mají maximální výšku danou počtem registrů, ovšem v lepším případě jsou zredukovány, což sníží počet testů registrů.

Nakonec je potřeba aktualizovat hodnoty registrů po provedení přechodu. Jelikož přiřazení do všech registrů probíhá při provedení přechodu RSA paralelně, nelze provádět přiřazení v libovolném pořadí. Mohla by nastat situace, kde je přepsána hodnota registru, která je později potřeba na pravé straně jiného přiřazení. Pro řešení této situace je z přiřazení v přechodu vytvořen graf závislostí registrů. V případě cyklů jsou některá přiřazení rozdělena na dvě přiřazením nové hodnoty registru r do jeho kopie r' a následným přiřazením $r \leftarrow r'$. Jakmile jsou cykly odstraněny, je vygenerováno topologické uspořádání, podle kterého lze již bezpečně přiřazení provádět.

Samotný VM matcher, naimplementovaný v C++, poté provádí vygenerovaný kód pro zadané vstupy. Provádění kódu prochází vygenerované instrukce a vykonává je. Současný stav RSA je ‘zakódován’ pouze pozicí v programu, stavy registrů jsou uloženy jako pole množinových datových struktur (konkrétně tabulek rozptýlených hodnot).

Naimplementovaný VM matcher, pojmenovaný **rsacpp**, byl experimentálně ověřen. Experimenty probíhaly jako simulované ReDoS útoky. Regexy pro tyto útoky byly získány z projektů z praxe a škodlivé vstupy k nim byly vygenerovány nástrojem RENGAR. Výsledky ukazují, že (i) **rsacpp** je o několik řádů rychlejší než předchozí prototyp co se týče rychlosti matchingu a několikanásobně rychlejší co se týče kompilace regexu, a (ii) zatímco ReDoS útoky způsobily extrémní zpomalení všech testovaných průmyslových matcherů (všechny z nich dosáhly v několika případech 100 s), **rsacpp** prokázal imunitu vůči všem testovaným ReDoS útokům a všechny zpracoval v čase pod hranicí 0,1 s. Toto ukazuje významný potenciál navržené techniky zvýšit odolnost veřejně nasazených aplikací vůči ReDoS útokům. Vědecký článek založený na této diplomové práci byl přijat k publikaci na mezinárodní konferenci PLDI'26 [22].

Advanced Approaches for Data Languages

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. Ing. Ondřej Lengál Ph.D. I have also worked with Ing. Vojtěch Havlena Ph.D. and doc. Ing. Lukáš Holík Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. The LLM Claude was used for proof-reading. ChatGPT was used during implementation to generate some boilerplate code and functions used for debugging.

.....
Jan Vašák
June 8, 2026

Acknowledgements

I would like to thank my supervisor, Ondra Lengál, for his guidance, feedback, and, most importantly, patience throughout the work on this thesis. I would also like to thank my colleagues and co-authors, Vojta Havlena and Lukáš Holík, for their collaboration on the related publication. Finally, I am so lucky that I have more names to list from my personal life than would be practical in this section. So, to avoid leaving anybody out, I finish with this general (and rather generic) sentence: I would like to thank my family, friends, and loved ones in general for their support during the writing of this thesis as well as during my studies.

Contents

1	Introduction	2
2	Automata for Data Words and Other Preliminaries	4
2.1	Register Automata	4
2.2	Register Set Automata	7
2.3	Other Automata Models	10
2.4	Graphs	11
2.5	Multi-terminal Binary Decision Diagrams	13
3	Matching Regexes with Back-references	16
3.1	Backtracking Matchers	17
3.2	Matching Based on Register Set Automata	18
3.3	Prototype Overview	21
3.4	Other Proposed Approaches	23
3.5	Unicode	23
4	Improvements to the Prototype	25
4.1	NRA Construction Optimizations	25
4.2	Determinisation Optimizations	26
4.3	Unsuccessful Optimizations	27
5	Virtual Machine Matcher Design and Implementation	29
5.1	Virtual Machine Matcher Design	29
5.2	Code Generation	31
5.3	Code Execution	37
6	Experiments	41
6.1	Experimental Setup	41
6.2	Evaluating Improvements to RSA-based Matching	42
6.3	Comparison with Other Matchers	43
7	Conclusion	45
	Bibliography	47
A	Contents of the Included Storage Media	51
B	PLDI Publication	52

Chapter 1

Introduction

Back-references are an extension of regular expressions (regexes) that allow a regex to capture a substring of the input and match the exact captured substring to the input again later. Back-references are supported by most major regex matchers and libraries [15, 18, 23, 29, 30, 34]. However, in all major regex matching tools, back-references are matched using backtracking algorithms. A backtracking algorithm matches a regex by constructing a non-deterministic automaton representing the regex and then trying to find a matching path in the automaton, backtracking upon failure. The time complexity for backtracking algorithms is exponential in the worst case. For this reason, regex matchers with a focus on performance, such as RE2 [19] and HyperScan [42], do not support back-references [17, 25].

In various online services, a regex matcher encountering a difficult matching problem can cause the server to become unresponsive. Such a situation, if caused intentionally by an attacker, is the so-called *regular expression denial of service attack* (ReDoS). ReDoS is not a theoretical threat, for example, a StackOverflow outage in 2016 and a vulnerability in the widely-used Express.js framework were both caused by a ReDoS attack [2, 37]. Crucially, any current matcher that supports back-references is inherently vulnerable to ReDoS, since backtracking is used.

Register set automata (RSA) are a recently introduced automata model capable of deterministically representing a class of regexes with back-references [22]. RSAs extend finite automata with a finite set of registers, where each register can store a set of input symbols. In this work, we additionally allow sets of symbols instead of individual symbols on the transition labels of RSAs, which makes regex representation easier. The representation works by converting the regex to a register automaton (RA). However, RAs are not determinisable in general, so RSAs are used to determinise them (which allows a significantly larger class of RAs to be determinised). The deterministic RSA representing a regex can then be simulated for an input word to perform regex matching. This simulation is, unlike in the case of backtracking, deterministic, i.e., there exists only one path in the RSA for a given input word.

This thesis introduces a new robust regex matcher capable of matching a class of regexes with back-references in linear time (for a fixed number of capture groups). The matcher is a significant improvement over a previous prototype [40]. It is a virtual machine (VM) matcher; that is, it interprets code representing an RSA on the regex matching input word. It is composed of two parts: the VM matcher itself and a code generator.

The above-mentioned prototype was improved in a major way in this work and the result is a significantly more efficient and robust regex matcher. The prototype was implemented in the Python programming language and works in the above-outlined manner: regex $\rightarrow$ [RA construction] $\rightarrow$ non-deterministic RA $\rightarrow$ [determinisation] $\rightarrow$ deterministic

RSA $\rightarrow$ [RSA simulation] $\rightarrow$ true/false result. Optimizations in this work mostly targeted RA construction and determinisation. During RA construction, the number of states and the number of registers of the constructed RA were reduced. Determinisation was optimized using techniques specific to RA constructed from regexes and as a result, the numbers of states and transitions of the resulting RSA were reduced.

The code generator designed in this thesis works on top of the improved prototype and begins working after determinisation is finished; i.e., code representing the constructed RSA is generated. To select the next transition based on the set of symbols on the transition labels, a prefix tree is generated with leaves representing the symbol sets. These prefix trees are defined outside of the code itself and are passed into the memory of the VM. For a set of symbols, multiple transitions differentiated by register tests can exist. To select a transition based on the contents of the registers, multi-terminal binary decision diagrams (MTBDDs) were implemented. The terminal nodes in these MTBDDs represent specific RSA transitions, and the non-terminal nodes represent register tests. Moreover, optimizations of the order of assignments to registers are introduced in order to reduce the numbers of times register values are copied.

The VM matcher itself, implemented in C++, executes the generated code for the regex matching inputs. Code execution consists of traversing the generated instructions and executing them. The current state of the RSA is ‘encoded’ as the position in the code and the register values are represented with an array of set data structures.

The implemented VM matcher, denoted **rsacpp**, was experimentally evaluated. The experiments were simulating ReDoS attacks on regexes from real-world projects. The malicious inputs were generated using the tool RENGAR [43]. The results show that (i) **rsacpp** is multiple orders of magnitude faster than the previous prototype in terms of matching speed and multiple times faster in terms of compilation speed, and (ii) while the ReDoS attacks caused extreme slowdown of all tested state-of-the-art matchers (all of the matchers had multiple instances reaching the 100 s time out), **rsacpp** proved to be immune against the ReDoS attacks, handling all of them in under 0.1 s. This proves significant potential of the designed technique to increase the resistance of publicly deployed applications against ReDoS attacks. An article based on this master’s thesis was accepted for publication at the international conference PLDI’26 [22].

The structure of the thesis is as follows. Chapter 2 defines register automata, register set automata, briefly describes other automata models for data words, and defines directed graphs and multi-terminal binary decision diagrams, which are models used in the implementation. Chapter 3 defines regexes with back-references, describes techniques for matching them, and provides an overview of the existing regex matcher prototype based on register set automata. Chapter 4 shows improvements made over the existing prototype in this work. Chapter 5 describes the C++ virtual machine matcher implemented in this work and Chapter 6 presents experiments made with both the improved prototype and the C++ matcher, comparing them with state-of-the-art regex matching tools. Finally, Chapter 7 summarizes the work in this thesis and provides discussion on it, including future work.

Chapter 2

Automata for Data Words and Other Preliminaries

This chapter introduces automata models for data words and other preliminaries. Specifically register automata and register set automata will be discussed in detail, as they are the key models in this work. Other related models will be discussed more briefly later in this chapter, and at the end of the chapter, graphs and binary decision diagrams will be introduced.

First, data words and other terminology will be formalized. $A \uplus B$ denotes a disjoint union of sets A, B (i.e., $A \cup B$ where $A \cap B = \emptyset$). Sometimes, $\cdot$ is used to denote an *ellipsis* (a value that can be ignored). For a function f and a subset of f 's domain X , $f|_X$ denotes a restriction of f to X . To denote the class of languages accepted by a model we will use the model's abbreviation, e.g. FA, short for finite automata, is how the set of regular languages would be denoted with this system.

An *alphabet* is a finite non-empty set. The elements of an alphabet are called *symbols*. A *word* of length n over an alphabet Σ is a sequence of symbols $a_1 a_2 \dots a_n$, where $\forall 1 \leq i \leq n: a_i \in \Sigma$. The set of all words over an alphabet Σ is denoted Σ^* and a *language* $\mathcal{L}$ over Σ is a set $\mathcal{L} \subseteq \Sigma^*$.

A *data domain* is a countably infinite set. The elements of a data domain are called *data values*. A data word of length n over an alphabet Σ and a data domain $\mathbb{D}$ is a sequence of pairs $(a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$, where $\forall 1 \leq i \leq n: (a_i, d_i) \in \Sigma \times \mathbb{D}$. The set of all words over an alphabet Σ and a data domain $\mathbb{D}$ can be denoted $(\Sigma \times \mathbb{D})^*$ and a *data language* $\mathcal{L}$ over $\Sigma \times \mathbb{D}$ is a set $\mathcal{L} \subseteq (\Sigma \times \mathbb{D})^*$. The empty (data) word is denoted ε , and the length of a (data) word w is denoted $|w|$.

2.1 Register Automata

Register automata were introduced by Kaminski and Francez in [26] (as finite-memory automata) as a model of computation dealing with infinite alphabets. They were intended to recognize an infinite-alphabet analog of regular languages. Register automata generalize finite automata by adding a set of registers, each of which can store a data value. The registers can then be tested for (non-)equality to the input value.

A (non-deterministic) *register automaton* over $\Sigma \times \mathbb{D}$ is a tuple $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$, where Q is a finite set of *states*, $I \subseteq Q$ is the set of *initial states*, $F \subseteq Q$ is the set of *final states*, $\mathbf{R}$ is a finite set of *registers*, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \rightarrow \mathbf{R} \uplus \{in, \perp\}) \times Q$

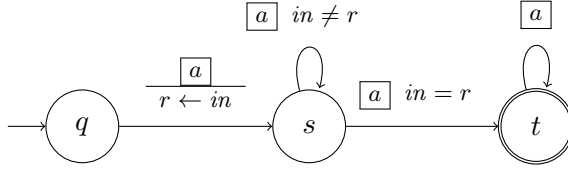


Figure 2.1: A register automaton accepting the language $\mathcal{L}_{first}^{repeat}$ of words whose first data value appears more than once

is a *transition relation* such that if $t = (q_1, a, g^-, g^\neq, up, q_2) \in \Delta$, then $g^- \cap g^\neq = \emptyset$. We denote t as $q_1 - \boxed{a \mid g^-, g^\neq, up} \rightarrow q_2$ ¹.

Intuitively, $\mathcal{A}$ can use t to move from q_1 to q_2 if it reads an input with the symbol a and a data value d such that d is equal to the value of all the registers in g^- and to none of the values of the registers in $g^\neq$. The values of registers are then updated according to up , where each register is (i) assigned a value of another register, or (ii) assigned d , denoted by in , or (iii) emptied, denoted by assigning $\perp$ to it. We will now describe this formally using *configurations* of $\mathcal{A}$.

A configuration of $\mathcal{A}$ is a pair $(q, c) \in Q \times (\mathbf{R} \rightarrow \mathbb{D} \uplus \{\perp\})$, where q is the current state and c is the current register assignment. An *initial configuration* of $\mathcal{A}$ is a pair $(i, \{r \mapsto \perp \mid r \in \mathbf{R}\})$, where $i \in I$. Given two configurations $\gamma_1 = (q_1, c_1)$ and $\gamma_2 = (q_2, c_2)$, $\mathcal{A}$ can make a *step* from γ_1 to γ_2 over $(a, d) \in \Sigma \times \mathbb{D}$ using $t = q_1 - \boxed{a \mid g^-, g^\neq, up} \rightarrow q_2 \in \Delta$, denoted as $\gamma_1 \vdash_t^{(a,d)} \gamma_2$ iff

1. $\forall r \in g^-: d = c_1(r)$,
2. $\forall r \in g^\neq: d \neq c_1(r)$, and
3. $\forall r \in \mathbf{R}: c_2(r) = \begin{cases} c_1(r') & \text{if } up(r) = r' \in \mathbf{R}, \\ d & \text{if } up(r) = in, \text{ and} \\ \perp & \text{if } up(r) = \perp. \end{cases}$

A *run* of $\mathcal{A}$ on a word $(a_1, d_1) \dots (a_n, d_n) \in (\Sigma \times \mathbb{D})^*$ is a sequence of configurations $\gamma_0 \gamma_1 \dots \gamma_n$ such that $\forall 1 \leq i \leq n: \exists t \in \Delta: \gamma_{i-1} \vdash_t^{(a_i, d_i)} \gamma_i$ and γ_0 is an initial configuration. A run $(q_0, c_0) \dots (q_n, c_n)$ of $\mathcal{A}$ on $w \in \Sigma \times \mathbb{D}$ is called an *accepting run* if $q_n \in F$.

A word w is *accepted* by $\mathcal{A}$, if there exists a run ρ of $\mathcal{A}$ on w such that ρ is an accepting run. The language accepted by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words that are accepted by $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma \times \mathbb{D} \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$.

$\mathcal{A}$ is *deterministic* if $|I| \leq 1$ and for every $q \in Q, a \in \Sigma$, it holds that for any two transitions $t_1 = q - \boxed{a \mid g_1^-, g_1^\neq, up_1} \rightarrow q'_1, t_2 = q - \boxed{a \mid g_2^-, g_2^\neq, up_2} \rightarrow q'_2 \in \Delta$ such that $t_1 \neq t_2$ we have $g_1^- \cap g_2^\neq = \emptyset$ or $g_1^\neq \cap g_2^- = \emptyset$. $\mathcal{A}$ is *complete* if $\forall q \in Q, a \in \Sigma, g \subseteq \mathbf{R}: \exists q - \boxed{a \mid g^-, g^\neq, \cdot} \rightarrow \cdot \in \Delta: g^- \subseteq g \wedge g^\neq \subseteq \mathbf{R} \setminus g$.

We will shorten register automata to RA, deterministic register automata to DRA. We sometimes use NRA if we want to highlight that an RA is non-deterministic (however, the meaning of NRA is the same as RA). We will also omit register updates of the form $r \leftarrow r$ and consider them implicit.

¹The transition notation was borrowed from [22].

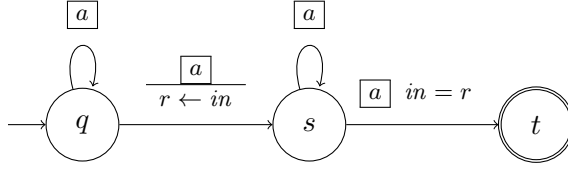


Figure 2.2: A register automaton accepting the language $\mathcal{L}_{last}^{repeat}$ of words whose last data value appears more than once

Example 2.1. Consider the language $\mathcal{L}_{first}^{repeat} = \{(a_0, d_0) \dots (a_n, d_n) \in (\Sigma \times \mathbb{D})^* \mid \exists 0 < i \leq n: d_i = d_0\}$, i.e., the language of words whose first data value appears more than once in the word. The automaton shown in Figure 2.1 accepts $\mathcal{L}_{first}^{repeat}$ for $\Sigma = \{a\}$. The automaton first stores the first data value into the register r while moving to state s . It then remains in s as long as the input data value is not equal to the data value in r . Once the data value of r is encountered in the input, the automaton moves to the accepting state t , where it remains until the end of the input word. Notice that this automaton is deterministic.

Example 2.2. Next, we examine the language $\mathcal{L}_{last}^{repeat} = \{(a_0, d_0) \dots (a_n, d_n) \in (\Sigma \times \mathbb{D})^* \mid \exists 0 \leq i < n: d_i = d_n\}$, i.e., the language of words whose last data value appears more than once in the word. An RA accepting this language for $\Sigma = \{a\}$ is shown in Figure 2.2. The automaton stays in the state q , ignoring the input data values, until it non-deterministically moves to state s , storing the input data value in the register r . In state s , the automaton loops while ignoring the input data values again, until it non-deterministically moves to state t . The move to state t is only possible if the input data value is equal to the value of r (if the values match, the automaton can non-deterministically choose either transition from state s). Notice that even though $\mathcal{L}_{first}^{repeat}$ and $\mathcal{L}_{last}^{repeat}$ appear similar, non-determinism is necessary to accept $\mathcal{L}_{last}^{repeat}$ with an RA, since, intuitively, it involves guessing what the last data value will be during the run.

2.1.1 Properties of Register Automata

We first examine the closure properties of RAs under Boolean operations. One of the original goals of RAs was to preserve as many of the closure properties of finite automata as possible.

Fact 2.1. [26, Theorem 3 and Remark 2] *RA is closed under union and intersection. RA is not closed under complement*

Proof idea. For closure under union and intersection, we can use the same product constructions as for finite automata. For non-closure under complement, consider the language $\mathcal{L}_{last}^{repeat} \in \text{RA}$ (discussed in Example 2.2). The complement of this language is $\mathcal{L}_{last}^{\neg repeat} = \{(a_0, d_0) \dots (a_n, d_n) \in \Sigma \times \mathbb{D} \mid \forall 0 \leq i < n: d_i \neq d_n\}$, that is, the language of words whose last data value appears exactly once in the word. $\mathcal{L}_{last}^{\neg repeat}$ is not accepted by any RA. If there were an RA $\mathcal{A}$ accepting $\mathcal{L}_{last}^{\neg repeat}$, $\mathcal{A}$ would need to store every data value in the word to check that none of them are equal to the last data value. Since the possible number of data values in a word is unbounded, $\mathcal{A}$ cannot exist and therefore RA is not closed under complement. $\square$

The non-closure under complement is an important property of the model in the context of this work. If the class is limited to deterministic automata, however, the closure holds even for complement.

Fact 2.2. [26, Section 4] *DRA is closed under union, intersection, and complement.*

Proof idea. For closure under union and intersection we can use the same construction as in the proof of Fact 2.1, as the construction preserves determinism. To obtain closure under complement, the DRA is transformed into a complete DRA and the finality of states is swapped. $\square$

Another important property of RA in the context of this work is that non-determinism makes RAs strictly more powerful. In other words, not every NRA can be determinised.

Fact 2.3. [26, Remark 2] *DRA $\subsetneq$ RA.*

Proof idea. Since RA is not closed under complement (Fact 2.1), while DRA is (Fact 2.2), there must be some languages in RA that are not in DRA. Trivially, all languages in DRA are also in RA. Therefore DRA $\subsetneq$ RA. $\square$

Lastly, we look at the emptiness and universality problems for RA. That is, the problems of deciding whether the language of a given RA is $\emptyset$ (emptiness) or $(\Sigma \times \mathbb{D})^*$ (universality). We omit the proofs of both of these properties here, since they are more involved.

Fact 2.4. [14, Theorems 4.3 and 5.1] *The emptiness problem for RA is PSPACE-complete.*

Fact 2.5. [7] *The universality problem for RA is undecidable.*

2.2 Register Set Automata

Register set automata were introduced in [22]. They generalize register automata by allowing registers to store sets of data values (as opposed to individual data values). As will be shown later, this allows them to represent a large class of NRAs deterministically, which is crucial for their application in regex matching.

A (non-deterministic) register set automaton is a tuple $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$, where Q , $\mathbf{R}$, I , and F are the same as for a register automaton, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \rightarrow 2^{\mathbf{R} \uplus \{in\}}) \times Q$ is a transition relation such that if $t = (q_1, a, g^\in, g^\neq, up, q_2) \in \Delta$, then $g^\in \cap g^\neq = \emptyset$. We denote t as $q_1 \xrightarrow{a \mid g^\in, g^\neq, up} q_2$.

Intuitively, $\mathcal{A}$ can use t to move from q_1 to q_2 if it reads an input with the symbol a and a data value d such that d is a member of all the registers in $g^\in$ and none of the registers in $g^\neq$. The values of the registers are then updated according to up , where each register is assigned a set of values, which can either be a register or in . Each register r is then updated by performing a union of all the registers in $up(r)$, and adding the input data value if $in \in up(r)$. As with RAs, we use configurations to define this formally.

A *configuration* of $\mathcal{A}$ is a tuple $(q, c) \in Q \times (\mathbf{R} \rightarrow 2^{\mathbb{D}})$, where q is the current state and c is the current register assignment. An *initial configuration* of $\mathcal{A}$ is a pair $(i, \{r \leftarrow \emptyset \mid r \in \mathbf{R}\})$, where $i \in I$. Given two configurations $\gamma_1 = (q_1, c_1)$ and $\gamma_2 = (q_2, c_2)$, $\mathcal{A}$ can make a *step* from γ_1 to γ_2 over $(a, d) \in \Sigma \times \mathbb{D}$ using transition $t = (q_1, a, g^\in, g^\neq, up, q_2) \in \Delta$, denoted as $\gamma_1 \vdash_t^{(a,d)} \gamma_2$, iff

1. $\forall r \in g^\in: d \in c_1(r)$,

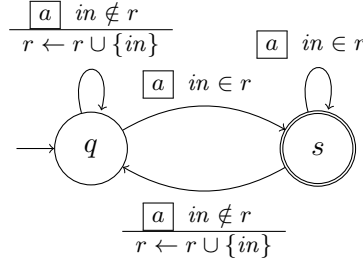


Figure 2.3: A deterministic register set automaton accepting the language $\mathcal{L}_{last}^{repeat}$ of words whose last data value appears more than once

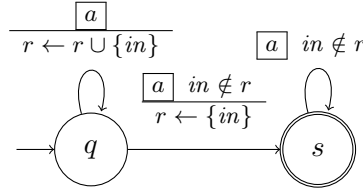


Figure 2.4: A register set automaton accepting the language $\mathcal{L}^{\neg \forall repeat}$ of words containing a data value exactly once

2. $\forall r \in g^\notin: d \notin c_1(r)$, and

3. $\forall r \in \mathbf{R}: c_2(r) = \bigcup \{c_1(r') \mid r' \in up(r) \wedge r' \in \mathbf{R}\} \cup \begin{cases} \{d\} & \text{if } in \in up(r), \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

A run of $\mathcal{A}$, accepting run of $\mathcal{A}$, and the language accepted by $\mathcal{A}$ have the same definitions as for RAs. $\mathcal{A}$ is *deterministic* if $|I| \leq 1$ and for every $q \in Q, a \in \Sigma$, it holds that for any two transitions $t_1 = q \xrightarrow{a \mid g_1^\in, g_1^\notin, up_1} q'_1, t_2 = q \xrightarrow{a \mid g_2^\in, g_2^\notin, up_2} q'_2 \in \Delta$ such that $t_1 \neq t_2$ we have $g_1^\in \cap g_2^\notin = \emptyset$ or $g_1^\notin \cap g_2^\in = \emptyset$. $\mathcal{A}$ is *complete* if $\forall q \in Q, a \in \Sigma, g \subseteq \mathbf{R}: \exists q \xrightarrow{a \mid g^\in, g^\notin, \cdot} \cdot \in \Delta: g^\in \subseteq g \wedge g^\notin \subseteq \mathbf{R} \setminus g$. We will shorten register set automata to RSA, deterministic register set automata to DRSA. Updates of the form $r \leftarrow \{r\}$ will be omitted and considered implicit.

Example 2.3. In this example, we revisit $\mathcal{L}_{last}^{repeat}$, which was discussed in Example 2.2. Figure 2.3 shows an RSA accepting $\mathcal{L}_{last}^{repeat}$ for $\Sigma = \{a\}$. The RSA works by accumulating all encountered data values into its register r and moving to accepting and non-accepting states based on whether the last-seen data value was new or not. Whenever the RSA sees a data value not yet stored in its register r , it moves to state q (non-accepting) and adds the new data value to r . If it sees a value already in r , it moves to the accepting state s . Observe that this RSA is *deterministic*, as opposed to the RA accepting the same language shown in Figure 2.2. In fact, this DRSA accepts a language no DRA can accept (as discussed in Facts 2.1 and 2.3).

Example 2.4. Consider the language $\mathcal{L}^{\neg \forall repeat} = \{(a_0, d_0) \dots (a_n, d_n) \in (\Sigma \times \mathbb{D})^* \mid \exists 0 \leq i \leq n: \forall 0 \leq j \leq n: i \neq j \implies d_i \neq d_j\}$, i.e., the language of words containing a data value with exactly one occurrence. Figure 2.4 shows an RSA accepting $\mathcal{L}^{\neg \forall repeat}$. The RSA works by looping in state q while accumulating all encountered data values into its register r .

When the RSA sees a value not stored in r , it can non-deterministically choose to move to state s while overwriting the contents of r with the input data value (intuitively, this is the data value the RSA has guessed to be unique in the word). In state s the RSA loops while checking that the input value is not in r (and is therefore different from the guessed value — making sure that the guessed value is unique in the word).

2.2.1 Properties of Register Set Automata

In the case of RSAs, we will first look at the emptiness and universality problems. Even though they generalize RAs, they retain decidability of the emptiness problem.

Fact 2.6. [22, Theorem 4.1] *The emptiness problem for RSAs is Ackermann-complete.*

Proof idea. Achieved by reducing to and from transfer Petri net coverability. $\square$

The Ackermann complexity class is a strict superset of all primitive-recursive classes. See, e.g., [36, Section 2.1.3] for details on the complexity class.

Fact 2.7. *The universality problem for RSAs is undecidable.*

Proof. RSAs generalize RAs, and RA universality is undecidable (Fact 2.5), therefore RSA universality is undecidable. $\square$

The closures under Boolean operations for RSA and DRSA remain the same as for RA and DRA respectively. Consequently, the relationship of RSA and DRSA also mirrors the relationship of RA and DRA.

Fact 2.8. [22, Theorem 4.2] *RSA is closed under union and intersection. RSA is not closed under complement.*

Proof idea. To show closure under union and intersection, we can use the same constructions as we used for RAs. To show non-closure under complement, consider again the language $\mathcal{L}^{\neg\forall repeat}$ (discussed in Example 2.4). It is possible to show that its complement, $\mathcal{L}^{\forall repeat} = \{(a_0, d_0) \dots (a_n, d_n) \in (\Sigma \times \mathbb{D})^* \mid \forall 0 \leq i \leq n \exists 0 \leq j \leq n: i \neq j \wedge d_i = d_j\}$ cannot be accepted by an RSA. If it could be accepted, then RSA could be used to decide the emptiness problem of a Minsky machine, which is undecidable. $\square$

Fact 2.9. [22, Theorem 4.4] *DRSA is closed under union, intersection, and complement.*

Proof. The constructions for union and intersections preserve determinism. To show closure under complement, any DRSA can be converted to a complete equivalent DRSA and then finality of its states can be swapped. $\square$

Fact 2.10. [22, Theorem 4.6] *DRSA $\subsetneq$ RSA.*

Proof idea. Consider the language $\mathcal{L}^{\neg\forall repeat}$, which is in RSA, but its complement is not. Since DRSA is closed under complement, $\mathcal{L}^{\neg\forall repeat}$ cannot be in DRSA. Trivially, every DRSA is also an RSA. $\square$

Lastly, we will examine the relationship between RAs and RSAs. We have already mentioned that RSAs generalize RAs. This generalization is indeed strict.

Fact 2.11. *RA $\subsetneq$ RSA.*

Proof idea. Clearly, we can use registers in RSAs the same way as registers in RAs by simply allowing storing at most one data value in them. Then (non-)membership tests are equivalent with (non-)equality tests. Therefore, $RA \subseteq RSA$. To show $RA \not\subseteq RSA$, consider the language $\mathcal{L}_{last}^{repeat}$, which we have shown to be accepted by both RAs and DRSA (Examples 2.2 and 2.3 respectively). However, $\mathcal{L}_{last}^{\neg repeat}$, the complement of $\mathcal{L}_{last}^{repeat}$, is not accepted by any RA (Fact 2.1), while it is accepted by a DRSA, since DRSA is closed under complement (Fact 2.9). $\square$

2.3 Other Automata Models

There are, of course, other models that work with data words. In this section, we will first discuss models similar to RSAs, and then mention a few other models briefly. The most similar to RSAs are *history register automata* introduced by Grigore and Tzevelekos in [20]. History register automata, like RSAs, have registers² storing sets of values. However, the update mechanism for history register automata only allows adding or removing the current input from the registers (i.e., history register automata do not allow register unions). History register automata have the same closure properties and decision problem complexities as RSA. In terms of expressive power, if we consider an extension of RSAs allowing removal of the input symbol, then every history register automaton has an equivalent RSA, but the other direction is an open problem. The deterministic variant of RSAs is proven to be strictly more powerful than deterministic history register automata. See [40, Chapter 4] for a detailed comparison of the two models.

Another model which resembles RSAs are set-augmented finite automata, introduced by Banerjee et al. in [3]. Set-augmented finite automata also have registers storing sets of values, but they are more restrictive than RSA. Namely, they only allow a singleton set as a guard (i.e., they only allow checking (non-)membership of one register on a transition), and they only allow updating one register at each transition by adding the input value (or alternatively no registers are updated). This model is also strictly less expressive than RSA, since (i) clearly we can construct an equivalent RSA for any set-augmented finite automaton, and (ii) set-augmented finite automata are incomparable to RAs, which are strictly weaker than RSA.

Although both history register automata and set augmented automata have registers storing sets of values, neither of them seem sufficient for RA determinisation, since they lack unions, which are used by the determinisation algorithm (see Section 3.2).

Data automata by Bojanczyk et al. [8] are composed of two components: a non-deterministic letter-to-letter word (i.e, not a data word) transducer (each transition reads and writes exactly one symbol) called the *base automaton*, and a word automaton called the *class automaton*. Acceptance is based on running the input through the base automaton, splitting the output into multiple sub-words based on data values (each sub-word contains all symbols, in order, of a data value appearing in the whole word), and testing acceptance of sub-words on the class automaton. The class automaton must accept every sub-word for the input to be accepted.

Class-memory automata by Björklund and Schwentick [6] use the notion of class-memory functions. Such functions assign to every data value the state of the automaton that

²The original history register automata have both (set-)registers as in RSAs (called histories) and (singleton-)registers as in RAs (called simply registers). For us, it is sufficient to consider a version of them with only histories, as the registers do not add to their expressive power. For consistency with the previous models, we call histories registers.

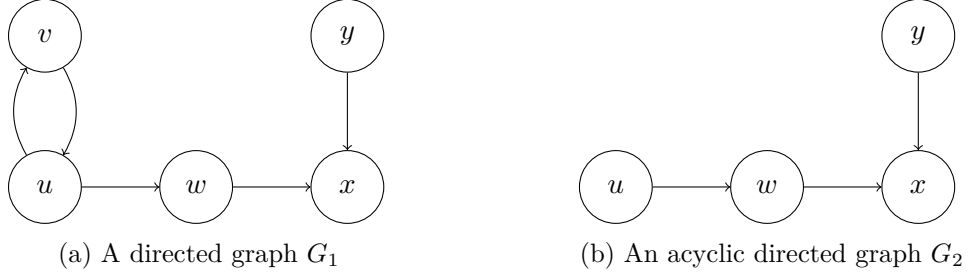


Figure 2.5: Examples of graphs

was assumed while reading the most recent position with the given data value. They are equivalent in expressive power to data automata. Class-memory automata are strictly more expressive than RA, but deterministic class-memory automata are incomparable to DRA. Their emptiness is decidable, but their universality is not (as they generalize RAs).

Pebble automata by Neven et al. [32] have a finite number of ‘pebbles’ that are used to mark positions in the input word. The automaton can compare data values lying under pebbles with the use of pebbles being restricted by a stack discipline (only the top pebble can be removed). Pebble automata are incomparable to RA and also incomparable to RSA [22]. The emptiness problem is undecidable for pebble automata.

2.4 Graphs

We briefly introduce graph theory and algorithms following [11]. A (directed) *graph* G is a tuple (V, E) , where V is a set of *vertices* (sometimes called nodes), and $E \subseteq V \times V$ is a set of *edges*. The outgoing degree $\deg(v)$ of a vertex v is defined $\deg(v) = |\{e \in E \mid e = (v, \cdot)\}|$. A *path* of length n in G is a sequence of vertices $v_1 \dots v_n$, such that $\forall 1 \leq i \leq n-1: (v_i, v_{i+1}) \in E$. A *cycle* of G is a path $v_1 \dots v_n$, where $n > 1$ and $v_1 = v_n$. G is *acyclic* iff no path of G is a cycle.

For two vertices $u, v \in V$ we say that v is *reachable* from u iff there exists a path $u \dots v$. A *strongly connected component* of G is a set of vertices $SCC \subseteq V$ such that $\forall (u, v) \in SCC \times SCC: v$ is reachable from u . Note that any strongly connected component SCC of G such that $|SCC| > 1$ must contain a cycle. I.e., there must exist a cycle $v_1 \dots v_n$ of G where $\forall 1 \leq i \leq n: v_i \in SCC$. It also holds that every vertex of a cycle must be in the same strongly connected component.

A *feedback vertex set* of G is a set FVS such that $G' = (V \setminus FVS, \{(u, v) \in E \mid u \notin FVS \wedge v \notin FVS\})$ is acyclic. The problem of finding a feedback vertex set of size $\leq k$ is NP-complete [27].

For a (directed) acyclic graph $G = (V, E)$, a *topological ordering* of G is a sequence of vertices $v_1 \dots v_n$, where $\{v_1, \dots, v_n\} = V$ and $\forall 1 \leq i \leq n: \forall i < j \leq n: v_i$ is not reachable from v_j .

Example 2.5. Figure 2.5 shows examples of directed graphs. Vertices are represented as circles and edges as arrows. Figure 2.5a shows a graph G_1 with a cycle uvu . The strongly connected components of G_1 are $\{u, v\}$, $\{w\}$, $\{x\}$, and $\{y\}$. There are two feedback vertex sets $\{u\}$ and $\{v\}$. The graph G_2 , shown in Figure 2.5b, was obtained by removing the feedback vertex set $\{v\}$ from G_1 . There are three different topological orderings of G_2 : ywx , $uywx$, and $uwyx$.

Algorithm 1: Depth-first search (DFS)

Input : A graph $G = (V, E)$ and a start vertex $s \in V$

Output: The set of vertices reachable from s

```
1 Function  $DFS\_visit(u)$ :  
2    $visited[u] \leftarrow 1$ ;  
3   foreach  $(u, v) \in E$  do  
4     if  $\neg visited[v]$  then  $DFS\_visit(v)$  ;  
5 foreach  $v \in V$  do  
6    $visited[v] \leftarrow 0$ ;  
7  $DFS\_visit(s)$ ;  
8 return  $\{v \in V \mid visited[v]\}$ ;
```

2.4.1 Algorithms

We examine algorithms for computing reachability, topological ordering, and strongly connected components. All complexity analyses will assume that the graphs are represented by an adjacency list (each node u has a list of vertices v , where $(u, v) \in E$).

Given a directed graph $G = (V, E)$ and a vertex $s \in V$, the set of vertices reachable from s in G can be found using a procedure called *depth-first search* (DFS). The procedure is shown in Algorithm 1. The core of DFS is in the recursive sub-procedure DFS_visit (Line 1). When called with a vertex u as an argument, it marks u as visited (Line 2). Then it calls itself recursively on every not yet visited v such that there is an edge from u to v (Line 4). This algorithm works in $\mathcal{O}(|V| + |E|)$ since every vertex is processed at most twice (once in initialization a once during the search) and every edge can be processed at most once.

To find a topological ordering of an acyclic graph, we can use a modified DFS. The algorithm is shown in Algorithm 2. We modify the DFS_visit procedure by prepending each processed vertex to a linked list (Line 5) to obtain the procedure DFS_visit_order (Line 1). The main procedure $topological_sort$ (Line 6) then makes sure that each vertex is processed by looping through the vertices and calling DFS_visit_order on each vertex that was not yet visited (Lines 10 to 11). This gives us a topological ordering, since a vertex u is only prepended after all vertices reachable from u are in the list already. This algorithm also works in $\mathcal{O}(|V| + |E|)$, since (i) every vertex is processed at once by DFS_visit_order and twice by $topological_sort$ and (ii) every edge is processed once by DFS_visit_order .

Algorithm 3 shows the computation of strongly connected components in a graph. We first abuse $topological_sort$ by calling it on a graph which may contain cycles (Line 1). Then, all edges in the graph are reversed (Line 2). In the reversed graph, we iterate over vertices based on the ordering (Line 7). For an unvisited vertex v , we obtain the list of nodes reachable from v in the reversed graph by clearing a list (Line 8) and calling DFS_visit_order from v with the empty list as an ordering (Line 9). The list now contains the vertices of a strongly connected component, which is appended to a list of strongly connected components (Line 10). This algorithm also works in $\mathcal{O}(|V| + |E|)$ as $topological_sort$ works in $\mathcal{O}(|V| + |E|)$, the edges can be reversed in $\mathcal{O}(|V| + |E|)$, and rest of the algorithm processes each vertex at most twice and each edge at most once.

Algorithm 2: Computing the topological ordering of an acyclic graph

Input : An acyclic graph $G = (V, E)$

Output: The topological ordering of G

```
1 Function DFS_visit_order( $u, ordering$ ):  
2    $visited[u] \leftarrow 1$ ;  
3   foreach  $(u, v) \in E$  do  
4     if  $\neg visited[v]$  then DFS_visit_order( $v$ ) ;  
5      $ordering.prepend(u)$ ;  
6 Function topological_sort( $G$ ):  
7   foreach  $v \in V$  do  
8      $visited[v] \leftarrow 0$ ;  
9    $ordering \leftarrow empty\_list()$ ;  
10  foreach  $v \in V$  do  
11    if  $\neg visited[v]$  then DFS_visit_order( $v$ ) ;  
12  return  $ordering$ ;
```

Algorithm 3: Computing the strongly connected components of a graph

Input : A graph $G = (V, E)$

Output: The strongly connected components of G

```
1  $ordering \leftarrow topological\_sort(G)$ ;  
2  $E^T \leftarrow \{(v, u) \mid (u, v) \in E\}$ ;  
3  $G \leftarrow (V, E^T)$ ;  
4  $sccs \leftarrow empty\_list()$ ;  
5 foreach  $v \in V$  do  
6    $visited[v] \leftarrow 0$ ;  
7 foreach  $v \in ordering$  do  
8    $scc \leftarrow empty\_list()$ ;  
9   if  $\neg visited[v]$  then DFS_visit_order( $v, scc$ );  
10   $sccs.append(scc)$ ;  
11 return  $sccs$ ;
```

2.5 Multi-terminal Binary Decision Diagrams

Binary decision diagrams (BDDs), introduced by Bryant in [10], are data structures used to represent Boolean functions. A Boolean function over n Boolean variables is a function with the following signature $f : \{0, 1\}^n \rightarrow \{0, 1\}$.

A *binary decision diagram* $\mathcal{B}$ is a tuple $(N, T, V, r, high, low, var, val)$, where N is a set of *non-terminal nodes*, T is a set of *terminal nodes* such that $N \cap T = \emptyset$, V is a finite set of *Boolean variables*, $r \in N \cup T$ is the *root node*, $high : N \rightarrow N \cup T$ defines the high successor, $low : N \rightarrow N \cup T$ defines the low successor, $var : N \rightarrow V$ assigns variables to non-terminal nodes, and $val : T \rightarrow \{0, 1\}$ assigns a Boolean value to each terminal node. The graph induced by $\mathcal{B}$ is defined as $G_{\mathcal{B}} = (V_G, E_G)$, where $V_G = N \cup T$, and $E = \{(x, y) \in V_G \times V_G \mid high(x) = y \vee low(x) = y\}$. It is required that $G_{\mathcal{B}}$ is acyclic and that $\forall v \in V_G : v$ is reachable from r . We say that $\mathcal{B}$ is *ordered* w.r.t. an ordering $\prec$ over V if it holds that $\forall n, m \in N : (m = high(n) \vee m = low(n)) \implies n \prec m$.

Algorithm 4: Ordered BDD reduction

Input : An ordered BDD $\mathcal{B} = (N, T, V, r, high, low, var, val)$

Output: A reduced ordered BDD $\mathcal{B}'$ equivalent to $\mathcal{B}$

```
1  $changed \leftarrow 1$ ;
2 while  $changed$  do
3    $changed \leftarrow 0$ ;
4   if  $\exists n \in N: high(n) = low(n)$  then
5      $changed \leftarrow 1$ ;
6      $\mathcal{B} \leftarrow replace\_node(\mathcal{B}, n, high(n))$ ;
7   if  $\exists n, n' \in N: high(n) = high(n') \wedge low(n) = low(n') \wedge var(n) = var(n')$  then
8      $changed \leftarrow 1$ ;
9      $\mathcal{B} \leftarrow replace\_node(\mathcal{B}, n, n')$ ;
10  if  $\exists t, t' \in T: val(t) = val(t')$  then
11     $changed \leftarrow 1$ ;
12     $\mathcal{B} \leftarrow replace\_node(\mathcal{B}, t, t')$ ;
13 return  $\mathcal{B}$ ;
```

W.l.o.g. we set $V = \{v_1, \dots, v_n\}$. For a node $x \in N \cup T$, the Boolean function $f_x: \{0, 1\}^n \rightarrow \{0, 1\}$ represented by x is defined as follows:

$$f_x(v_1, \dots, v_n) = \begin{cases} val(x) & \text{if } x \in T \\ (\neg var(x) \wedge f_{low(x)}(v_1, \dots, v_n)) \vee (var(x) \wedge f_{high(x)}(v_1, \dots, v_n)) & \text{if } x \in N \end{cases}$$

Two BDDs $\mathcal{B}_1 = (N_1, T_1, V, r_1, high_1, low_1, var_1, val_1)$ and $\mathcal{B}_2 = (N_2, T_2, V, r_2, high_2, low_2, var_2, val_2)$ are isomorphic iff there exists a bijection $f: (N_1 \cup T_1) \rightarrow (N_2 \cup T_2)$ such that (i) $\forall n \in N_1: f(n) \in N_2$ (and therefore $\forall t \in T_1: f(t) \in T_2$), (ii) $f(r_1) = r_2$, (iii) $\forall t \in T_1: val_1(t) = val_2(f(t))$, and (iv) $\forall n \in N_1: var_1(n) = var_2(f(n)) \wedge f(high_1(n)) = high_2(f(n)) \wedge f(low_1(n)) = low_2(f(n))$. A sub-BDD of $\mathcal{B}$ from node $n \in N \cup T$ is a BDD $\mathcal{B}_n = (N_n, T_n, V, n, high_{|N_n \cup T_n}, low_{|N_n \cup T_n}, var_{|N_n \cup T_n}, val_{|N_n \cup T_n})$, where $N_n = \{x \in N \mid x \text{ is reachable from } n \text{ in } G_{\mathcal{B}}\}$, $T_n = \{x \in T \mid x \text{ is reachable from } n \text{ in } G_{\mathcal{B}}\}$.

$\mathcal{B}$ is *reduced* if (i) $\forall n \in N: high(n) \neq low(n)$, and (ii) there are no two distinct nodes $n_1, n_2 \in N \cup T$ such that $\mathcal{B}_{n_1}$ and $\mathcal{B}_{n_2}$ are isomorphic. For two nodes $n, n' \in N \cup T$ of $\mathcal{B}$, we define the procedure $replace_node(\mathcal{B}, n, n') = (N', T', V, r', reroute_n(high), reroute_n(low), var_{|N'}, val_{|T'})$, where $N' = N \setminus \{n\}$, $T' = T \setminus \{n\}$, $reroute_n(f) = \{x \mapsto f(x) \mid x \in N \wedge f(x) \neq n\} \cup \{x \mapsto n' \mid x \in N \wedge f(x) = n\}$, and if $r = n$, then $r' = n'$ else $r' = r$. Algorithm 4 shows how an ordered BDD can be converted to an equivalent reduced ordered BDD. Line 6 removes non-terminal nodes whose high edge points to the same node as the low edge. Line 9 merges together non-terminal nodes that have the same variable and the same low and high descendants. Line 12 merges terminal nodes with the same values.

Example 2.6. Figure 2.6 shows an example of an ordered BDD over the variables a, b, c , where $a \prec b \prec c$. In the diagrams, non-terminal nodes are labeled with their respective variables, terminal nodes with their values, high edges are shown as solid lines, and low edges are shown as dashed lines. Figure 2.6a shows the BDD as fully branching (i.e., with the maximum possible number of nodes and edges given the variables). Figure 2.6b shows an equivalent BDD in its reduced ordered form.

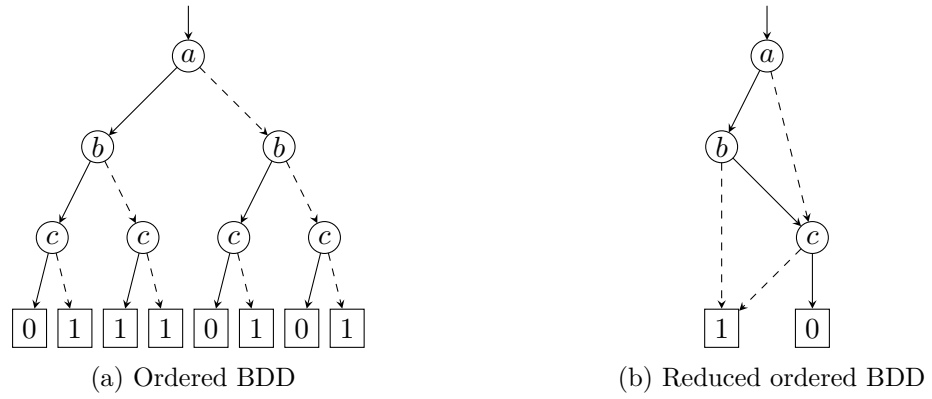


Figure 2.6: An ordered BDD over the variables a, b, c and its reduced ordered equivalent

In this work, we will use an extension of BDDs called *multi-terminal* BDDs (MTBDDs). They are the same as BDDs, except the values of terminals are not restricted to Boolean values. We can keep the same definition for ordered and reduced variants as well as keeping the reduction algorithm the same.

Chapter 3

Matching Regexes with Back-references

In this chapter, we will introduce regexes with back-references and common ways to match them. We will then present a method for matching a subset of regexes with back-references deterministically using register set automata.

Regexes with back-references extend standard regular expressions by capture groups and back-references. Intuitively, a capture group is a sub-expression that can be back-referenced, and back-references then match exactly the text matched by the capture group. We will define the syntax of back-references formally and then describe their semantics only informally, since the formalization can get quite technical and the semantics are not consistent across different implementations anyway. See, e.g., [5] for a precise formalization.

A *regex with back-references* (REBR) over an alphabet Σ is defined recursively as follows:

- (i) $\forall a \in \Sigma \cup \{\varepsilon\} : a$ is a REBR,
- (ii) $\forall n \in \mathbb{N} : \backslash n$ is a REBR,
- (iii) if R_1, R_2 are REBRs, then $(R_1 R_2)$, $(R_1 | R_2)$, (R_1^*) are also all REBRs, and
- (iv) if R is a REBR, then $\forall n \in \mathbb{N} : ({}_n R)_n$ is a REBR.

If we only consider items (i) and (iii), we get the definition for standard regexes, for which, given a regex R , we define the language of R as:

$$\mathcal{L}(R) = \begin{cases} \{R\} & \text{if } R \in \Sigma \cup \{\varepsilon\}, \\ \{vw \mid v \in \mathcal{L}(R_1) \wedge w \in \mathcal{L}(R_2)\} & \text{if } R = (R_1 R_2), \\ \{w \mid w \in \mathcal{L}(R_1) \vee w \in \mathcal{L}(R_2)\} & \text{if } R = (R_1 | R_2), \\ \{w_1 \dots w_n \mid n \in \mathbb{N}, \forall 1 \leq i \leq n : w_i \in \mathcal{L}(R_1) \cup \{\varepsilon\}\} & \text{if } R = (R_1^*), \end{cases}$$

Item (ii) represents back-references, and item (iv) capture groups. We mark back-references and capture groups with numbers to show which back-references reference which capture groups explicitly (in practice, it is usual that every pair of parentheses is considered a capture group and a back-reference $\backslash n$ references the n -th pair of parentheses). In a REBR, a back-reference must appear after the corresponding capture group, and there can be multiple back-references referencing the same capture group. In general, a capture group may contain any REBR.

We assign the regex operators their usual priorities, where iteration takes precedence over concatenation, which in turn takes precedence over branching. Parentheses that follow

this operator priority can then be omitted. For a REBR R , we will use R^+ as a shorthand for RR^* , $[a_1a_2 \dots a_n]$ as a shorthand for $(a_1|a_2| \dots |a_n)$, where $a_i \in \Sigma$. We lift this notation to sets of symbols, where $[S]$ denotes $[a_1a_2 \dots a_n]$, where $\{a_1, a_2, \dots, a_n\} = S, S \subseteq \Sigma$. We use $.$ as a shorthand for $[\Sigma]$. Additionally, $[\wedge a_1a_2 \dots a_n]$ denotes $[(\Sigma \setminus \{a_1, a_2, \dots, a_n\})]$.

Example 3.1. Consider the REBR $(_1.^*)_1 \backslash 1$. This REBR matches words which can be split down the middle into two equal subwords. I.e., it matches the language $\{ww \mid w \in \Sigma^*\}$. Notice that because back-references match exactly the subword matched by the capture group, REBRs can match non-regular languages, as demonstrated by this example. In fact, matching REBRs in general is NP-complete [1].

Example 3.2. Next, we show a different way to utilize back-references. The REBR $(_1.^+)_1 \backslash 1^+$ can be used to match words whose length is a composite number. That is, it matches the language $\{w \in \Sigma^* \mid \exists 1 < k, l < |w| : k \cdot l = |w|\}$. Intuitively, the capture group $(_1.^+)_1$ can be thought of as guessing k (it matches any word that is at least 2 characters long), and the number of repeats in the second part, $\backslash 1^+$, can be thought of as guessing $l - 1$.

Example 3.3. In this example we show how back-reference semantics can get complicated. Consider the REBR $(ab|(_1cd)_1) \backslash 1$. Clearly, it matches the word $cdcd$, but what about the word ab ? Usually, the answer is *no*, since the back-reference's corresponding capture group was not matched, however in JavaScript [18], for example, the answer is *yes*, since the back-reference is ignored if the corresponding capture group was not matched.

Example 3.4. Lastly, we further refine the intuitive definition we gave for REBR semantics. We stated that a back-reference matches exactly the subword matched by the capture group. A capture group, however, can potentially match multiple different subwords. Consider the REBR $(_1.^*)_1 \backslash 1$. The capture group can be matched any number of times (including zero, which corresponds to Example 3.3). It is more accurate to say that a back-reference matches exactly the subword *most recently matched* by the capture group. The example REBR $(_1.^*)_1 \backslash 1$ matches the word $abcc$, but not the word $abcb$. In general, it matches any word in which the last 2 symbols are the same.

3.1 Backtracking Matchers

In industrial matchers, REBRs, if supported, are matched using backtracking algorithms [15, 18, 23, 29, 30, 34]. This section will briefly introduce a general backtracking algorithm for matching regexes.

The first step of the algorithm is to convert the input REBR into some kind of *extended* non-deterministic finite automaton (ENFA). As part of the extension of finite automata, we assume a way to store a subword matched by a capture group and use it when checking a back-reference. The matching algorithm is shown in Algorithm 5.

The algorithm as shown is independent of the regex features supported, as it defers regex semantics to the underlying ENFA using the $make_step(\mathcal{A}, q, a)$ procedure. This procedure returns all the possible states an ENFA $\mathcal{A}$ can be in after making a step over the symbol a from state q . Algorithm 5 is recursive. It first checks if the input word w is empty, and if so, it checks whether the current state is final (Line 2). If there are remaining symbols in w , the algorithm takes the first one, and collects all the states the ENFA can be in after making a step from the current state over that symbol (Line 5). Then, for each

Algorithm 5: *match_backtrack*($\mathcal{A}, w, q_0$)

Input : Word w and an ENFA with $\mathcal{A}$ initial state q_0

Output: *true* if $w \in \mathcal{L}(\mathcal{A})$, else *false*

```
1 if  $w = \varepsilon$  then
2   | return is_final( $q_0$ );
3 else
4   |  $a \leftarrow w[0]$ ;
5   |  $S \leftarrow \text{make\_step}(\mathcal{A}, q_0, a)$ ;
6   | foreach  $q \in S$  do
7     | if match_backtrack( $\mathcal{A}, w[1 \dots], q$ ) then return true;
8 return false;
```

state collected, the algorithm recursively calls itself from that state with the remainder of the input word (Line 7). If any matching path inside the ENFA is found, the algorithm returns *true*, otherwise it falls through to Line 8 and returns *false*.

Algorithm 5 has an exponential worst-case complexity (in the length of the input), since for a bad regex and input combination, it has to explore all possible paths in the ENFA, branching at each step. Consider, for example, the regex $(a^*)^*$ and a word $a^n b$. At each a , the backtracking engine can either continue matching a^* (the inner $*$) or start a new repetition of (a^*) (the outer $*$). Therefore each a introduces two possible branches the algorithm can take. And since the b at the end causes the match to fail, the algorithm will backtrack through all of these possibilities, which gives us the exponential complexity. For this reason, performance focused regex matchers, such as RE2 [19] and HyperScan [42], avoid backtracking algorithms (and do not support back-references) [17, 25].

3.2 Matching Based on Register Set Automata

In this section, we introduce REBR matching using register set automata as presented in [22] with refinements from [40]. It is possible to use RSAs to match a subset of REBRs. The high-level algorithm consists of three steps:

1. Convert the REBR into an NRA (if possible).
2. Determinise the NRA from the previous step into a DRSA (if possible).
3. Run the input word on the DRSA from the previous step.

When using RSAs to match REBRs, we only consider *single-letter* REBRs, that is, REBRs whose capture groups are of length one. Note that it is possible to convert any REBR with bounded-length capture groups into single-letter REBRs by splitting the longer capture groups into a concatenation sequence of single-letter ones (and splitting the back-references accordingly)¹. E.g., $(1 \dots)_1 \backslash 1$ turns into $(1 \dots)_1 (2 \dots)_2 (3 \dots)_3 \backslash 1 \backslash 2 \backslash 3$.

For the purposes of REBR matching in practice, we use RAs and RSAs that run just on words over the alphabet, not on data words (or, equivalently, we set $\Sigma = \mathbb{D}$). Since we would still like to support an infinite (or very large) alphabet, we modify RAs and RSAs to have sets of symbols on their labels instead of just singular symbols. To convert a single-letter REBR into an NRA we use a modified Thompson’s construction [39]. The modification

¹However, single-letter REBRs obtained this way usually turn out to be non-determinisable into DRSA.



Figure 3.1: Automata created in the extended Thompson's construction for a capture group $(_n[S])_n$ and back-reference $\backslash n$

consists of the two additional rules, one for capture groups, and one for back-references. The capture group rule for the n -th capture group is an automaton with a transition that stores the input symbol in register r_n , the transition is labeled with the set of symbols the REBR allows to match in the capture group. The back-reference automaton for the n -th back-reference simply tests whether the input symbol is a member of the register r_n . The transition label is always Σ in this case. Figure 3.1 shows the automata for both new rules.

Next, we show the determinisation of an NRA into a DRSA from [22] (with modifications from [40]). We start by introducing some notation. Let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be an NRA. The set $\mathbf{R}[q]$ denotes the set of registers *active* in $q \in Q$, that is, $\mathbf{R}[q]$ is the set of registers for which there exists a transition $q_1 \xrightarrow{a \mid g^=, g^\neq, up} q_2 \in \Delta$, where (i) $q_2 = q$ and $up(r) \neq \perp$ or (ii) $q_1 = q$ and $r \in g^= \cup g^\neq$. For a set of registers $S \subseteq Q$ we denote $\mathbf{R}[S] = \bigcup_{q \in S} \mathbf{R}[q]$. We say that $\mathcal{A}$ is *register-local*, if for any two states $q_1, q_2 \in Q$ s.t. $q_1 \neq q_2$ it holds that $\mathbf{R}[q_1] \cap \mathbf{R}[q_2] = \emptyset$. $\mathcal{A}$ is *relation-free* if for each transition it holds that each register appears on the right-hand side of an update at most once, and at most one register can be updated by *in* or a register in the equality guard. In other words, $\mathcal{A}$ is relation-free if for any input word $w = (a_0, d_0) \dots (a_n, d_n)$ and position i , the data value d_i is stored in at most one register (note that values equal to d_i might be stored in multiple registers, provided that each of them was read from a different position in w). Every NRA can be modified to be register-local and relation-free.

Algorithm 6 shows the determinisation algorithm. The algorithm's basis is the subset construction for finite automata with special handling of registers. Instead of macrostates being just sets of states, they are pairs $(S, c) \in 2^Q \times (\mathbf{R} \rightarrow \{0, 1, \omega\})$, where S is a set of states (same as in the standard subset construction) and c is a mapping of registers to track whether they store more than one value. This is important because a non-membership test of a DRSA does not simulate a non-equality test of an NRA if the register stores more than one value, which is reflected in Line 9.

The algorithm uses a worklist to store macrostates it reaches, creating transitions and target macrostates from the currently processed macrostate. For a macrostate being processed, we first create the set M of *minterns* from sets of symbols on the transitions outgoing from each state in the set of states of our macrostate (Line 5). Intuitively, a minterm is a region of a Venn diagram drawn from the given sets. In other words, a member of the set M is a set of symbols that are not distinguished between by transition of the NRA, whose origin state is a member of S . For each combination of $a \in M$ and a subset g of non-empty registers active in the current macrostate (Line 6) the algorithm creates a set of transitions that can be taken using (a, d) , assuming that d is stored in all registers in g (Line 7). Lines 10 to 16 then process the collected transitions and a new transition simulating them is created (Line 25). Lines 17 to 20 check that the generated DRSA transition does not overapproximate the language of the original NRA by checking that each

Algorithm 6: Determinisation of an NRA into a DRSA

Input : Relation-free, register-local 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 $\perp$

```

1  $\mathcal{Q}' \leftarrow \text{worklist} \leftarrow I' \leftarrow \{(I, c_0 = \{r \mapsto 0 \mid r \in \mathbf{R}\})\};$ 
2  $\Delta' \leftarrow \emptyset;$ 
3 while  $\text{worklist} \neq \emptyset$  do
4    $(S, c) \leftarrow \text{worklist.pop}();$ 
5    $M \leftarrow \text{minterms}(\Delta, S);$ 
6   foreach  $a \in M, g \subseteq \{r \in \mathbf{R}[S] \mid c(r) \neq 0\}$  do
7      $T \leftarrow \{q - \boxed{a \mid g^-, g^\neq, \cdot} \rightarrow q' \in \Delta \mid q \in S, g^- \subseteq g, g^\neq \cap g = \emptyset\};$ 
8      $S' \leftarrow \{q' \mid \cdot - \boxed{\cdot \mid \cdot, \cdot, \cdot} \rightarrow q' \in T\};$ 
9     if  $\exists q - \boxed{\cdot \mid \cdot, g^\neq, \cdot} \rightarrow q' \in T, \exists r \in g^\neq: c(r) > 1$  then return  $\perp$  ;
10     $T^\bullet = \{q - \boxed{a \mid g^-, g^\neq, \text{up}[g^-/\text{in}]} \rightarrow q' \mid q - \boxed{a \mid g^-, g^\neq, \text{up}} \rightarrow q' \in T\};$ 
11    foreach  $r_i \in \mathbf{R}$  do
12       $\text{tmp} \leftarrow \emptyset;$ 
13      foreach  $\cdot - \boxed{\cdot \mid g^-, \cdot, \text{up}} \rightarrow \cdot \in T^\bullet$  do
14        if  $\text{up}(r_i) = y \neq \perp \wedge c(y) \neq 0$  then  $\text{tmp} \leftarrow \text{tmp} \cup \{y\}$  ;
15       $\text{op}_{r_i} \leftarrow \begin{cases} \text{tmp} \setminus \{\text{in}\} & \text{if } \text{tmp} \cap g \neq \emptyset \text{ and } \\ \text{tmp} & \text{otherwise} \end{cases};$ 
16       $c'(r_i) \leftarrow \sum_{x \in \text{op}_{r_i}}^{>1 \rightsquigarrow \omega} c(x)$  ;
17    foreach  $q' \in S'$  do
18       $P \leftarrow \text{op}_{r_1} \times \dots \times \text{op}_{r_n}$  for  $\{r_1, \dots, r_n\} = \mathbf{R}[q'];$ 
19      foreach  $(x_1, \dots, x_n) \in P$  do
20        if  $\nexists (\cdot - \boxed{\cdot \mid \cdot, \cdot, \text{up}} \rightarrow q') \in T^\bullet$  s.t.  $\bigwedge_{1 \leq i \leq n} \text{up}(r_i) = x_i$  then return  $\perp$  ;
21       $\text{up}' \leftarrow \{r_i \mapsto \text{op}_{r_i} \mid r_i \in \mathbf{R}\};$ 
22      if  $(S', c') \notin \mathcal{Q}'$  then
23         $\text{worklist.push}((S', c'));$ 
24         $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{(S', c')\};$ 
25       $\Delta' \leftarrow \Delta' \cup \{(S, c) - \boxed{a \mid g, \mathbf{R} \setminus g, \text{up}'} \rightarrow (S', c')\};$ 
26 return  $\mathcal{A}' = (\mathcal{Q}', \mathbf{R}, \Delta', I', \{(S, c) \in \mathcal{Q}' \mid S \cap F \neq \emptyset\});$ 

```

register update combination is also present in some of the transitions of the NRA that is being simulated.

Since this overapproximation check ignores the actual values of registers, if there exists a pair of registers that hold equal value regardless of input, the algorithm can detect overapproximation even when the language of the resulting DRSA does not actually overapproximate the NRA's language. Notably, this often happens with copies of the same register created to satisfy register-locality. This overapproximation check is refined by using the post-processing algorithm of [40], which keeps track of which registers are copies of one another.

3.2.1 Matching Complexity

The actual matching process is running the input word on the created DRSA. The complexity of computing a run on a word is dependent on the implementation. Consider the

registers being implemented by a sparse set, a data structure introduced in [9]. A sparse set consists of two arrays, *sparse* and *dense*. The *dense* array is an array of the ordinal values of each element of the set in the order that they are added into the set. The *sparse* array is an array where for each element's ordinal value i , $sparse[i]$ is set to the index at which i is stored in *dense*. Additionally, the number of elements in the set is stored.

This means that one does not have to initialise memory used by neither *sparse* nor *dense*, since the valid values of *dense* are determined by the number of elements in the set, and the valid values of *sparse* are determined by being pointed to from *dense*. The reason that this data structure was chosen here is that we get membership testing and element adding at constant time complexity, and a union of two sets in time complexity linear in the number of elements of the smaller of the two sets.

Note that this data structure is impossible to create over an infinite domain. However, this is not a big problem, since we only need to represent as many symbols as there are in the given input word (which is a finite amount). Also, in practice, the alphabet is actually finite. For a finite alphabet, this means that an update can be performed in $\mathcal{O}(|\Sigma| \cdot |\mathbf{R}|^2)$.

To decide which transition to take, we need to test each register at most once. This can be done by, for example, storing the transitions as leaves of a decision tree structure, with membership of the input in a register determining the next node at the level corresponding to the register. This therefore has a time complexity of $\mathcal{O}(|\mathbf{R}|)$. Combining everything, we get the following theorem.

Theorem 3.1. [22, Theorem 6.3] *For a fixed DRSA over a finite alphabet, the worst-case time complexity of regex matching is linear to the length of the input word w , specifically $\mathcal{O}(|w| \cdot |\Sigma| \cdot |\mathbf{R}|^2)$.*

Note that if the alphabet is infinite, the complexity turns into $\mathcal{O}(|w|^2 \cdot |\mathbf{R}|^2)$. However, it is possible to define a *copy-free* property on DRSA's. A DRSA is copy-free if it has no transition where it assigns the same non-singleton register to two registers.

Theorem 3.2. [22, Theorem 6.4] *For a copy-free DRSA, the complexity of testing membership is $\mathcal{O}(|w| \cdot |\mathbf{R}|)$.*

Proof idea. The proof is done by ordering input values based on age (the oldest one being the one read earliest) and setting a register's age as the age of the oldest value stored inside it. We can perform each necessary union by adding the values from the younger register to the older register. Since the automaton is copy-free, a value from a specific position in the input word can only be added to another register $|\mathbf{R}|$ times (after $|\mathbf{R}|$ unions, it is in the oldest register and cannot be added anywhere else). This means that each position of an input word can be processed at most $\mathcal{O}(|\mathbf{R}|)$ times, giving us an overall matching complexity of $\mathcal{O}(|w| \cdot |\mathbf{R}|)$. $\square$

3.3 Prototype Overview

The prototype is implemented in Python. For the most part, it adheres to the procedures outlined in Section 3.2. It uses Python's regex parser to create a syntax tree from a REBR. NRAs are created using Thompson's construction with the added rules shown in Figure 3.1. The transition labels are sets of symbols; they are represented as tuples (neg, S) , where neg is a Boolean value and S is an explicit set of characters. Semantically, the value of (neg, S) is S if neg is false, and $\Sigma \setminus S$ if neg is true. For example, $\{a, b, c\}$ is represented

as $(false, \{a, b, c\})$, $\Sigma \setminus \{a, b, c\}$ as $(true, \{a, b, c\})$, Σ is represented by $(true, \emptyset)$, and $\emptyset$ as $(false, \emptyset)$. Standard set operations are implemented for this tuple representation. The NRA itself is represented closely according to the theory as a class containing sets of states, final states, initial states, transitions and registers.

Note that during NRA construction, syntax tree nodes that are not supported might be encountered, in which case the construction fails. This includes back-referenced capture groups which are longer than one character, look-arounds (including any non-character consuming anchors, such as the word boundary, commonly represented as $\backslash b$). The exceptions to this are the start and end anchors — $\wedge$ and $\$$ — which cause the matcher to omit the implicit $.^*$ at the start or end of a REBR, which is added to any unanchored REBR. Any matching flags (such as case-insensitive or multiline matching) are ignored. Consequently, the prototype matcher does not take any special care when reading newline characters. If line by line matching is required, it is necessary to feed the matcher one line at a time.

Once an NRA is constructed, ε -transitions are removed from the NRA using a standard ε -closure algorithm. Since states that are potentially unreachable may be generated, they are removed using graph reachability. Note that technically there could remain some state which is unreachable due to some register condition, for example, a membership test of a register which is always empty at that point. We do not deal with these, since removing unreachable states in general solves NRA emptiness, which is PSPACE-complete.

A constructed NRA is converted to a register-local form. First, implicit register updates of the form $r \leftarrow r$ are filled in if r is unspecified (i.e., not assigned any value) in the given update (the NRA does not initially have these, since it simplifies Thompson’s construction). The conversion algorithm iterates over transitions, modifying each update and test by prepending the respective state names to the register names. After the register is in its register-local form, in each given update, each unspecified register r gets the update $r \leftarrow \perp$ so it only remains active in the correct states. See Figure 3.2 for an example of the result of the register-local conversion (for clarity, states are shown in the superscript instead of prepended).

The resulting automata are relation-free without modification since, intuitively, different capture groups (and back-references to them) do not interact in single-letter REBRs. Alternatively, we can inspect the rules for the automaton construction that manipulate registers (other than just keeping their value the same) shown in Figure 3.1, and see that no relations or non-equality guards can be introduced in them.

The determinisation implementation also closely follows Algorithm 6. The resulting DRSA is also represented as a class with sets of states, final states, initial states, transitions and registers during the determinisation. At the end of the determinisation algorithm, a hash table is created mapping macro states to their outgoing transitions. The method for running a word on a DRSA keeps track of the current state and current values of registers. The transition to be taken is found using the hash map and iterating over the transitions returned, until one is possible to take given the state and register configuration. The first transition found can safely be taken since the automaton is deterministic. If no transition is found, the method returns false. The register configuration is updated by processing register updates one by one, creating a new configuration (while the current one still has old values). After all register updates on a transition are processed, the new configuration replaces the old one. If there are no more input symbols to be read, the method returns true if the current state is in the set of final states and false otherwise.

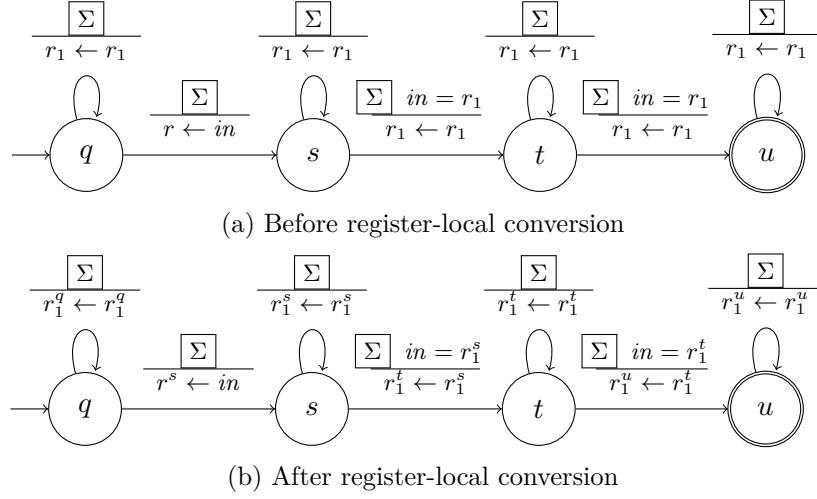


Figure 3.2: NRA for the REBR $.*(.)_1.*\backslash 1.*$ before and after conversion to its register local form.

3.4 Other Proposed Approaches

This section will discuss approaches other than the standard backtracking, that have been proposed for matching REBRs.

In [16], Freydenberger and Schmid introduce a notion of deterministic REBRs by restricting REBRs using a deterministic automaton model. For practical matching though, this class is too restrictive, since it does not cover regular languages (it is incomparable with the class of regular languages). In [35], Schmid presents a subclass of REBRs that can be matched in polynomial time. Yang, et al. propose an algorithm for matching REBRs in [44]. The experimental results show that their approach algorithm is more resistant to ReDoS attacks than the standard backtracking approach, however, the approach retains the exponential worst-case complexity (as back-references are supported in full). Nogami and Terauchi show an algorithm for matching REBRs with one (general) capture group and one back-reference that works in quadratic-time [33]. This is incomparable to RSA-based regex matching, which can support multiple capture groups, but only of length at most one.

3.5 Unicode

We will briefly introduce Unicode and its UTF-8 (Unicode Transformation Format 8-bit) encoding specifically. The Unicode Standard [38] is a standard for encoding written characters and text, supporting written languages from all over the world. Each character is assigned a *code point* (a numeric value) and a name. There are 1,114,112 code points in unicode, not all of which are assigned a character. Unicode is built on top of ASCII such that ASCII characters have assigned code points with the same value as their ASCII encoding.

The UTF-8 encoding is the most common Unicode encoding, as evidenced by the fact that 99% of websites are encoded using UTF-8 [41]. Encoding a code point into UTF-8 takes one to four bytes of space. An advantage of UTF-8 is that ASCII characters are encoded exactly the same as in ASCII and that it is the most compact encoding for text that is mostly (extended) ASCII. The number of bytes in an encoding is determined by the prefix of the first byte: ‘0’ for one byte, ‘110’ for two bytes, ‘1110’ for three bytes,

Table 3.1: UTF-8 Encoding (Table 3-6 in [38])

Scalar Value	1st Byte	2nd Byte	3rd Byte	4th Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

and ‘11110’ for four bytes. Bytes which are not first in the encoding have the prefix ‘10’. Table 3.1 shows conversion of code point into its UTF-8 encoding. A *scalar value* is the binary representation of a code point.

Chapter 4

Improvements to the Prototype

This chapter discusses improvements made to the existing Python prototype. Excluding bug fixes, the improvements consist of extending parsing to support more REBRs and optimizations to the NRA construction and the determinisation algorithm. At the end of the chapter, some optimizations which were unsuccessful are discussed.

Parsing was modified to allow capture groups of length 0 and capture groups that have variable length between 0 and 1. Capture groups of length 0 are handled by generating both their capture group and back-reference transitions as ε -transitions. When a capture group of variable length 0–1 is encountered, the REBR is transformed into a union of two copies of the original REBR with one of the copies having the variable capture group be set to length 0 and the other to length 1. If there are multiple variable-length capture groups present in the REBR, each combination needs to be generated. E.g., the REBR $(1.\varepsilon)_1(2.\varepsilon)_2\backslash 1\backslash 2$ becomes $(1\varepsilon)_1(2\varepsilon)_2\backslash 1\backslash 2 \mid (3\varepsilon)_3(4.)_4\backslash 3\backslash 4 \mid (5.)_5(6\varepsilon)_6\backslash 5\backslash 6 \mid (7.)_7(8.)_8\backslash 7\backslash 8$ (white space is added for clarity).

4.1 NRA Construction Optimizations

Optimizations during NRA construction were made to reduce the number of states and the number of registers. The NRA Thompson’s construction and the ε -removal algorithm were adjusted to reduce the number of states of the resulting NRA. To reduce the number of registers, an analysis finding unnecessary registers was implemented

Reducing the Number of States. In the NRA Thompson’s construction, the concatenation was adjusted to detect whether its argument NRAs only contain ε -transitions. If one NRA only contains ε -transitions, the other argument NRA can be used as the concatenation directly. If both argument NRAs only contain ε -transitions, a new automaton with only one state, which is both final and initial, and no transition is returned to represent the concatenation.

In the ε -removal algorithm, an additional special case was implemented where if a state is detected with its only outgoing transition being an ε -transition, the state is deleted along with the outgoing ε -transition. All the transitions incoming into the deleted state are redirected to the target state of the deleted ε -transition. If the deleted state was initial, the target state of the deleted ε -transition is also made initial. The state and transition removal is not done if the detected state is final, since setting the target state of the ε -transition as final could change the language accepted by the NRA.

Reducing the Number of Registers. After the NRA is converted to its register-local form, there may be registers that can be deleted without any effect on the language of the NRA. Consider, e.g., the NRA shown in Figure 3.2b. It has 4 registers (r_1^q, r_1^s, r_1^t , and r_1^u), however, we could delete the registers r_1^q and r_1^u without any effect on the language accepted by the NRA.

An algorithm resembling data-flow analysis was implemented to detect dependencies between registers. In $dep: \mathbf{R} \rightarrow 2^{\mathbf{R}}$, for each register, the algorithm maps a set of registers that the register is dependent on, initially set to $\emptyset$ for all registers. Then, until dep remains unchanged, the updates up of each transition of the NRA are looped through and for all registers r , if $up(r) \in \mathbf{R}$, then dep is updated as follows: $dep(r) \leftarrow dep(r) \cup \{up(r)\} \cup dep(up(r))$. I.e., for each register r we obtain the set of registers whose value may be eventually copied into r .

After performing this dependency analysis, we remove all registers which are (i) not *tested* (a register is tested if any transition contains it in g^- or $g^\neq$) and (ii) not in $dep(r)$ for some tested register r . This not only helps speed up determinisation by removing registers, it also tightens overapproximation detection. The overapproximation detection may otherwise be triggered by an update to a register that the optimization would remove.

4.2 Determinisation Optimizations

There were two optimizations made to the determinisation algorithm (Algorithm 6). The first optimization leverages the fact that there are no non-equality guards in NRAs constructed from REBRs. This is because back-references only check for equality. Since there are no non-equality guards, we do not need to track the sizes of registers in the determinisation. We still keep track of which registers are empty, since this knowledge limits the number of possible register subsets created on Line 7, which turned out to be a large benefit. We therefore limit the number of macrostates created while not increasing the number of transitions generated per macrostate (which would be the case had we completely removed size tracking). This is done by collapsing all values > 1 to 1 instead of ω on Line 16.

The second optimization also reduces the number of register subsets created on Line 6. For a given (currently processed) macrostate (S, c) , we will call registers that appear as (non-)membership guards of on Line 25 *considered*. We collect all outgoing transitions from the states in S , and create a set of non-empty registers used on the (non-)equality guards. Then, only registers of the created set are considered instead of all registers. We do this by adding a line $\mathbf{R}_S \leftarrow \{r \in g^- \cup g^\neq \mid q - (a \mid g^-, g^\neq, \cdot) \rightarrow \cdot \in \Delta, \text{ where } q \in S\}$ ¹ between Lines 5 and 6 and replacing $\mathbf{R}$ with $\mathbf{R}_S$ on Lines 6 and 25. This is safe to do since testing any registers not used in the NRA transitions will not have an effect on the destination state or register update of the newly created DRSA transition. Limiting the number of registers considered was effective, since the number of outgoing transitions from a macrostate grows exponentially with the registered considered.

¹In the reduction of registers considered as well as the register dependency analysis we include $g^\neq$ to keep the optimizations general, however, in the implementation this is not needed since no non-equality guards occur in NRAs created from REBRs.

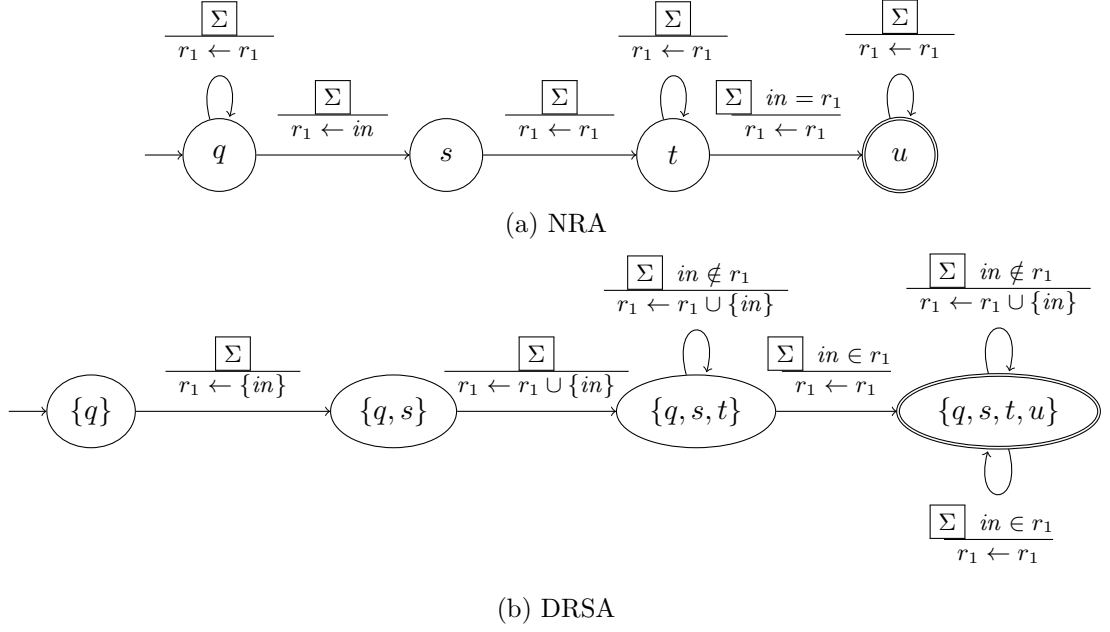


Figure 4.1: NRA and DRSA for the REBR $.*(1.)_1.^+\backslash 1.*$ if the automata were constructed without register localization

4.3 Unsuccessful Optimizations

This section will list optimizations to the prototype that were attempted and deemed to be unsuccessful. It was conjectured that register localization is not necessary for determinisation in the case of NRAs constructed from REBRs. Thus, register localization was removed at one point to reduce the number of registers in the automata. This was effective at reducing both determinisation time and matching time. However, it turns out that even in the case of NRAs created from REBRs, it is necessary for correctness to perform register localization before determinisation.

Consider the regex $.*(1.)_1.^+\backslash 1.*$, its NRA representation without register localization is shown in Figure 4.1a. The resulting DRSA of the determinisation algorithm (without overapproximation checking) running on the NRA is shown in Figure 4.1b. Note that Figure 4.1b omits the register mapping part of macrostates, the state $\{q\}$ should also be labeled with $\{r_1 \mapsto 0\}$ and the remaining states with $\{r_1 \mapsto 1\}$. The DRSA overapproximates the language of the NRA, for example, the word *abb* is accepted by the DRSA, but is not accepted by the original NRA (nor does it match the original REBR).

The next optimization was done in an attempt to speed up determinisation: minterms were constructed for all transitions of the NRA at the beginning, instead of for each macrostate's outgoing transitions. However, the matching time slowed down enough that this idea was scrapped.

Yet another optimization was implemented to speed up matching: the way registers are tested to select the next transition was modified. Instead of looping through all transitions per state and testing each transition's guards separately, registers were all tested when entering a state and a bitmap of membership was created, which was compared against each transition's guards. After this optimization was implemented, matching time was actually slower, so it was also scrapped.

Lastly, to decrease matching time using a lower level programming language, a different approach was first taken to using C++ for than is described in Chapter 5. Originally, each DRSA was compiled into C++ code matching it. More specifically, state control was realized as a big switch statement. To select a transition by its label a static hash set was tested for membership and to select a transition by register tests the transitions had an assigned bitvector of registers which need to contain the input for the transition to be available. A bitvector of membership of the current input symbol was created at each step to compare against the transitions' bitvectors. Transitions from a state were looped through sequentially and the first transition satisfying both conditions was taken. However, the generated code was typically very large and compilation was extremely slow.

Chapter 5

Virtual Machine Matcher Design and Implementation

The approach selected to implement an efficient RSA-based matcher was a virtual machine (VM) implemented in C++ and a code generator built on top of the existing prototype. The code generator generates code representing a DRSA for a given REBR and the VM efficiently performs matching by interpreting the code over an input string. This, of course, involves designing a code shared by both parts.

5.1 Virtual Machine Matcher Design

This section will present some design questions, give high-level answers to them and present the virtual machine code designed to work with the design choices.

Integral to performing a run on a DRSA is keeping track of its current configuration consisting of a state and a register assignment. Inspired by other matchers using a VM approach (for deterministic NFA simulation in their case) [4, 12], the current state of the DRSA is not explicitly represented. It is implicitly represented by a label in the code under which all actions performed in the state are encoded. The register assignment is represented explicitly as an array of sets. The actions performed for a state s are the following:

1. If s is final and there are no more input symbols left then accept the input.
2. Read the input symbol (reject the input if there are no more input symbols).
3. Decide the next transition to be taken.
4. If no transition was found then reject the input.
5. Update the registers based on the selected transition.
6. Go to the target state of the selected transition.

Most of these are straightforward to implement, the main two design questions address items 3 and 5.

1. Given a state of a DRSA, a register assignment, and an input symbol, how to efficiently determine the next transition (if any) to be taken?
2. Given a transition, how to efficiently perform its register update?

Note that the first question is actually a two part problem, since the transitions must be evaluated in terms of their labels as well as their register guards.

Labels. To select transitions based on their labels, we took inspiration from the RE2 regex matcher’s DFA implementation [19]. For an automaton running on ASCII, from each structure representing a state, there is an array of 256 pointers to next states for each possible byte value. However, we would like matchers run on Unicode, specifically the UTF-8 encoding, which is up to four bytes per character. The solution RE2 has for this is to embed the UTF-8 decoding of the character into the automaton itself. Our matcher does not embed the decoding into the automaton explicitly, but we use a similar approach to construct a tree that classifies, byte by byte, which symbol set (that is a transition label) the input character belongs to and decodes the character in the process. Such trees are computed during code generation for each state and not stored in the code directly, but passed to the matcher separately as data structures in an array indexed by states. The array is loaded into the matcher’s memory at initialization.

Register guards. To distinguish between different transitions with the same symbol set, register tests need to be handled. This is done by what is essentially an if-else chain in the code, where the registers are tested in a given order, each at most once. By the end of the chain, we can determine which transition the DRSA should take since every possible combination of register membership of the input value was explored. We can look at this if-else chain as a decision tree. If possible, it is desirable to reduce the tree to lower the number of register tests performed. To achieve this, multi-terminal binary decision diagrams were employed, where variables are registers, non-terminal nodes are specific register tests with jumps, and terminals are references to transitions to be performed.

Register updates. Since in a DRSA, register updates take place in parallel, we cannot simply assign registers in an arbitrary order, since we could overwrite a value of a register that is needed in another update. The simple solution to this is to compute the new value of each register into a new copy, and only when all register updates are computed, replace the registers with their respective new copies (recall that this approach is used in the Python prototype). Since we would like to avoid creating new register copies when not necessary, the approach we take with the VM matcher is to create a graph of dependencies between register updates, only create register copies to break cycles in the graph, and use topological ordering to order the register updates correctly.

5.1.1 Virtual Machine Instruction Set

To keep track of a configuration, the VM stores the instruction pointer ip , the current input symbol in , and a register array $\mathbf{R}_{\mathbf{VM}}$ holding the values of all registers (potentially including auxiliary registers not defined by the DRSA). The VM also has an array $\mathbf{mem}$, which stores the decoding trees for each state; this array is indexed by states. Table 5.1 shows the instruction set. The data types used as parameters of instructions are the following:

- `reg_id`— index into the $\mathbf{R}_{\mathbf{VM}}$ array,
- `state_id`— index into the $\mathbf{mem}$ array,
- `label`— a label in the code, and

Table 5.1: The instruction set of the VM matcher

Instruction	Parameters	Semantics
ACCEPT	—	If there are no symbols left in the input then accept.
FAIL	—	Reject the input.
JUMP	label l	Set ip to the index of the instruction labeled with l .
TEST	reg_id r , label l	If $in \in \mathbf{R}_{\mathbf{VM}}[r]$ then jump to l .
CLEAR	reg_id r	$\mathbf{R}_{\mathbf{VM}}[r] \leftarrow \emptyset$
ADDIN	reg_id r	$\mathbf{R}_{\mathbf{VM}}[r] \leftarrow \mathbf{R}_{\mathbf{VM}}[r] \cup \{in\}$
SWAP	reg_id r_1 , reg_id r_2	Swap the values of $\mathbf{R}_{\mathbf{VM}}[r_1]$ and $\mathbf{R}_{\mathbf{VM}}[r_2]$.
UPDATE	reg_id r , {reg_id} S	$\mathbf{R}_{\mathbf{VM}}[r] \leftarrow \mathbf{R}_{\mathbf{VM}}[r] \cup \bigcup_{A \in S} A$
DECODE	state_id s	If there are no symbols left in the input then reject. Otherwise jump to the label obtained by traversing the decoding tree $\text{mem}[s]$ and store the decoded input symbol into in .

- {reg_id}— a sequence of indices into the $\mathbf{R}_{\mathbf{VM}}$ array.

5.2 Code Generation

The code generation was implemented on top of the Python prototype. After building a DRSA from a REBR, the code and memory representing the DRSA is generated for the actual VM matcher to interpret.

The high-level algorithm for code generation is shown in Algorithm 7. First, the states are put into a list such that the initial state is first in the list (Line 1). This is done so that the execution can simply start at the very beginning of the program. For each state s , a label is generated (Line 3). The *statelabel* function is designed to return a unique state label for a given state such that it does not collide with other labels or instruction names. In the implementation, it simply returns the string ‘state_’ followed by the state’s id. If s is accepting, output the ACCEPT instruction (Line 4). Line 5 stores all transitions outgoing from s into Δ_s . If Δ_s is empty, we output a FAIL instruction and end the processing of s , since no transitions can be taken (Line 7). In the other case, a DECODE instruction is output for s (Line 9). Then, on Line 10, all transition labels (which are sets of symbols) of the transitions in Δ_s are collected into Σ_s .

Along with s , Σ_s is passed to the *generate_decoding_tree* function on Line 11. We will examine decoding tree generation in Section 5.2.1. For each set A in Σ_s , a jump label is output (Line 13) and instructions handling transitions from Δ_s with A as their label will be generated. The label created by *setlabel* will be the target of jumps made from the DECODE instruction generated for s . The created label must not collide with any other labels or instructions. In the implementation, it outputs the id of s and the id of A (which, in the implementation, is generated during the construction of Σ_s) with an underscore between them. Such transitions are collected into Δ_s^A (Line 14). Line 15 then stores all unique

Algorithm 7: Virtual machine code generation for a DRSA

Input : A DRSA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$

Output: Arrays **prg** and **mem** representing the generated machine code and memory respectively

```

1   $state\_order \leftarrow list(I) + list(Q \setminus I);$ 
2  foreach  $s \in state\_order$  do
3       $prg.append(statelabel(s));$ 
4      if  $s \in F$  then  $prg.append(ACCEPT);$ 
5       $\Delta_s \leftarrow \{t \in \Delta \mid t = s - \boxed{\cdot \mid \cdot, \cdot, \cdot} \rightarrow \cdot\};$ 
6      if  $\Delta_s = \emptyset$  then
7           $prg.append(FAIL);$ 
8          continue;
9       $prg.append(DECODE, s);$ 
10      $\Sigma_s \leftarrow \{A \mid s - \boxed{A \mid \cdot, \cdot, \cdot} \rightarrow \cdot \in \Delta_s\};$ 
11      $mem[s] \leftarrow generate\_decode\_tree(\Sigma_s, s);$ 
12     foreach  $A \in \Sigma_s$  do
13          $prg.append(setlabel(s, A));$ 
14          $\Delta_s^A \leftarrow \{t \in \Delta_s \mid t = s - \boxed{A \mid \cdot, \cdot, \cdot} \rightarrow \cdot\};$ 
15          $U \leftarrow \{(up, q) \mid s - \boxed{A \mid \cdot, \cdot, up} \rightarrow q \in \Delta_s\};$ 
16          $\mathcal{B} \leftarrow create\_MTBDD(\Delta_s^A, U);$ 
17          $generate\_MTBDD\_code(prg, \mathcal{B}, s, A);$ 
18         foreach  $(up, q) \in U$  do
19              $prg.append(update\_label(s, A, (up, q)));$ 
20              $up' \leftarrow order\_updates(up);$ 
21              $generate\_reg\_updates(up');$ 
22              $prg.append(JUMP, statelabel(q));$ 
23 return  $prg, mem;$ 

```

update and target state combinations from Δ_s^A into U . Using sets Δ_s^A and U , an MTBDD $\mathcal{B}$ is created and $\mathcal{B}$'s code is generated (Lines 16 and 17). MTBDD creation and code generation is discussed in Section 5.2.2. $\mathcal{B}$ is used to generate register tests and ultimately to jump to the correct update and target state combination. The labels that are potential targets of this jump are generated on Line 19. The function *update_label* is tasked with generating a unique label given the arguments. In the implementation, this label consists of the ids of s , A , and (up, q) joined by underscores; the id of (up, q) is generated when building the implementation's equivalent of U . Then, the update is ordered so that it is semantically correct, and the instructions executing the updates are generated (Lines 20 and 21). The algorithms for ordering updates and generating their code are discussed in Section 5.2.3. Lastly, a JUMP instruction to the destination state is output (Line 22).

5.2.1 Decoding Trees

Decoding trees are essentially prefix trees classifying UTF-8 encodings of symbols into minterm character sets of a given state. Therefore each internal node has 256 children, and each leaf has a code label. Recall the internal set representation as an explicit set and a negation Boolean value (*neg*, S). The function *generate_decoding_tree* starts with an empty tree, then for each set it adds all symbols from the explicit set one by one. A symbol

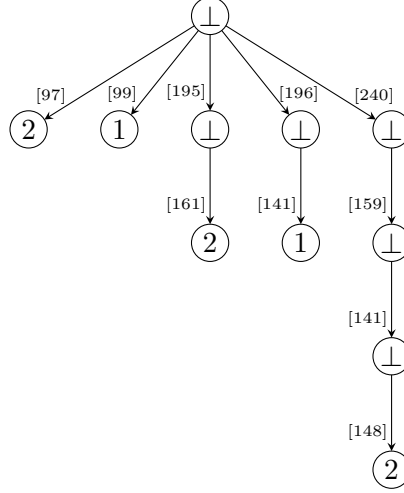


Figure 5.1: Decoding tree for the sets $\{c, \check{c}\}$ and $\{a, \acute{a}, \text{🍔}\}$

is first encoded into UTF-8 and then it is added by traversing the tree using the encoding, creating a new node whenever a node is missing (i.e., the parent has a null value at the corresponding index in the children array). Leaves do not get assigned labels yet, instead each minterm set has a bitmap of leaves which belong to it (pointers to leaves are stored in an array with indices corresponding to the bitmaps).

After all symbols have been added, the labels of each set are assigned to the corresponding leaves. The labels need to be specific to a state (and correspond to the labels generated on Line 13 in Algorithm 7), which is why *generate_decoding_tree* also takes a state as its argument. If the set is negative, all the leaves that have zeroes in the set’s bitmap need to have the set’s label added. This also involves looking up any null pointers in children arrays and creating leaves for them. Note that since the sets are minterms, they are disjoint and therefore at most one set will need to fill in the null pointers.

For cases where the only minterm is the set Σ , whose representation is $(true, \emptyset)$, there are no symbols added to the tree using this approach and the tree remains empty. We would therefore reach a null pointer while decoding, which normally means that the symbol is not part of any of the sets. We handle this by adding a special label for this case. The meaning of this label is that no jumps should be made (as all transitions fall under the same minterm and their code will be generated right below the **DECODE** instruction).

Example 5.1. Figure 5.1 shows an example of a decoding tree for the sets $\{c, \check{c}\}$ with label ‘1’ and $\{a, \acute{a}, \text{🍔}\}$ ¹ with label ‘2’. Nodes are labeled by their label field. Edges are marked with the index the corresponding child is stored at. Only non-null children are shown. The byte values of the UTF-8 encodings of the mentioned characters are as follows: c – [99], $\check{c}$ – [196, 141], a – [97], $\acute{a}$ – [195, 161], 🍔 – [240, 159, 141, 148].

5.2.2 MTBDD Construction

MTBDDs are used to generate register tests, which find an appropriate transition to execute. The terminal values are elements of U , and variables are registers with an arbitrary ordering that is fixed across the entire DRSA code generation. Non-terminal nodes are

¹The hamburger emoji is manually ‘typeset’ as a png from the OpenMoji project [21]

represented as a structure storing their variable, low and high successors, and an array of their parents (other nodes that point to this node with either their high or low edge).

MTBDD Creation. The initial MTBDD is created by looping through Δ_s^A and adding each transition as a path. The path corresponding to a transition $s - \boxed{A \mid g^\epsilon, g^\zeta, up} \rightarrow q$ is the one where variables of registers in g^ϵ are set to 1, variables of registers in g^ζ are set to 0, and the terminal node of the path is the id of (up, q) . The paths are added in a similar manner to the approach taken with decoding trees (see Section 5.2.1). The MTBDD is traversed for as long as the nodes exist and then any missing nodes needed to complete the path are created.

Note that not all registers need to be in g^ϵ or g^ζ , but for each transition in Δ_s^A , $g^\epsilon \cup g^\zeta$ is the same. I.e., $\exists S \subseteq R: \forall \cdot - \boxed{\cdot \mid g^\epsilon, g^\zeta, \cdot} \rightarrow \cdot \in \Delta_s^A: g^\epsilon \cup g^\zeta = S$. This is due to a modification made to the determinisation algorithm discussed in Section 4.2.

MTBDD Reduction. After each transition has been added, the MTBDD is reduced. The reduction algorithm essentially follows the theoretical one shown in Algorithm 4. It runs two sub-procedures that (i) merge duplicate nodes (corresponding to Line 9) and (ii) remove nodes with no branching (corresponding to Line 6). The sub-procedures are run repeatedly, in the given order, until neither of them modifies the MTBDD.

The sub-procedure (i) creates a hash table of non-terminal nodes using their variable and their high and low successors (specifically the object ids of the successors) as keys. The nodes are added to the hash table in a bottom-up order, which ensures that children of any added node are already non-duplicates. If a node's key is already stored in the hash table, it gets replaced with the node from the hash table, changing the parents of the replaced node to point to the hash table node, and adjusting the hash table node's parents array. Terminal nodes are hashed already during MTBDD creation using their value as a key. Therefore they are already non-duplicate and do not need to be handled in this procedure.

The sub-procedure (ii) goes through the nodes top-down, deleting any node whose high and low fields point to the same child node. The parents of the deleted node are adjusted to point to the child node and added to the child node's parents array. If the deleted node was the root of the MTBDD, the child node is set as the new root.

MTBDD Code Generation. For a reduced MTBDD, the code for the register tests and eventual jump to the correct transition label is generated. If the root of the MTBDD is a terminal node, or a non-terminal node with no successors, no code needs to be generated. Otherwise we will process the nodes recursively. For a node n , starting with the root, we perform the following operations:

1. If visiting n for the first time, generate a jump label for it. In the implementation the jump label is of the form of a string '*prefix_node_node_id*', where *prefix* is the jump label generated on Line 13 of Algorithm 7 and *node_id* is n 's id.
2. Generate an instruction **TEST** *reg label*, where *reg* is $var(n)$, and *label* is the jump label of $high(n)$.
3. Generate an instruction **JUMP** *label*, where *label* is the jump label of $low(n)$.
4. Call this recursively on n 's non-terminal successors.

The jump labels for terminal nodes are the ones generated on Line 19 in Algorithm 7.

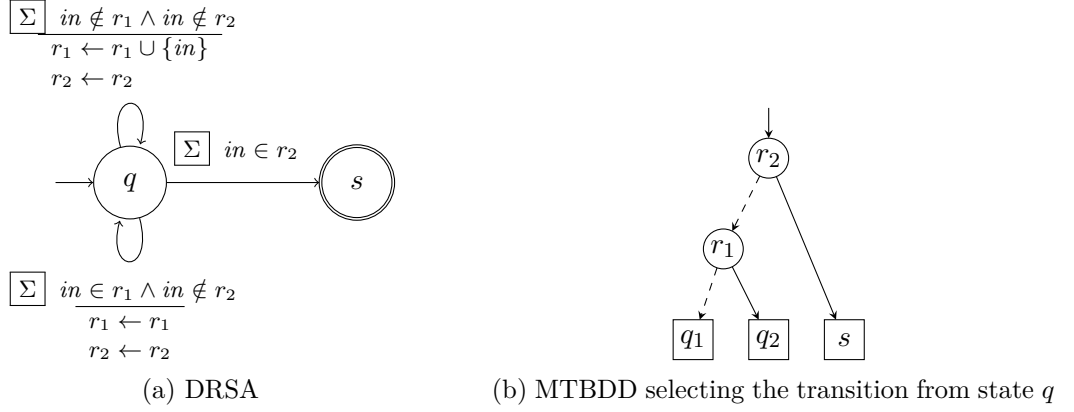


Figure 5.2: A DRSA and an MTBDD selecting a transition from the DRSA's state

Example 5.2. Consider the DRSA shown in Figure 5.2a. The DRSA is arbitrarily chosen to show a simple example of an MTBDD selecting a transition from one of its states. Figure 5.2b shows an MTBDD selecting a transition from the state q . Since there is only one symbol set (Σ) present on the transition label of the DRSA, the next transition is entirely decided by the MTBDD. The terminals each represent a unique destination state and update combination. In this case, each terminal represents a single transition with q_1 being the transition from q to q that is depicted above q , q_2 being the transition from q to q depicted below q , and s being the transition from q to s . Non-terminals represent a membership test of the register they are labeled with, with the high edge indicating the result of the the test being true.

5.2.3 Update Ordering

For a given update of all registers, we want to generate instructions that execute it. We start by building a graph $G = (V, E)$ from the update up (over $\mathbf{R}$), where $V = \mathbf{R}$, $E = \{(r, y) \mid r \in \mathbf{R} \wedge y \in up(r) \wedge y \neq in \wedge y \neq r\}$. An edge (r_1, r_2) represents that the update of r_1 is dependent on the old value of r_2 , and therefore r_2 cannot be updated before r_1 is. Self-loops are excluded from G because we do not mind a dependency of a register on its own old value. In order to use topological ordering to obtain the safe order of register updates, we must ensure that G is acyclic.

Intuitively, we will split updates whose nodes are part of a cycle into two. An update $r \leftarrow up(r)$ will be split into $r' \leftarrow up(r)$ and $r \leftarrow r'$. In G , this means that outgoing edges of r are transformed into outgoing edges of r' . There are no incoming edges added to r' , since it is an artificial register and no other register values are dependent on its old value. To preserve the relationship of r and r' we will use a bijection mapping them to one another.

The algorithm we use to remove cycles is shown in Algorithm 8. It starts by constructing a list of strongly connected components (Line 1). The function $SCCs(G)$ returns a list of SCCs, each represented as a set of its nodes. We check whether the G has a cycle on Line 2 by checking whether the number of SCCs is lower than the number of vertices. Since finding the minimal set of vertices to remove to make the graph acyclic is NP-hard (see Section 2.4), we use a heuristic: in each SCC we select the vertex with the highest outgoing degree to be split (Line 3). We proceed with splitting the selected nodes by creating ids for their copies (Line 5), mapping the original nodes to the copies in a bijection mapping

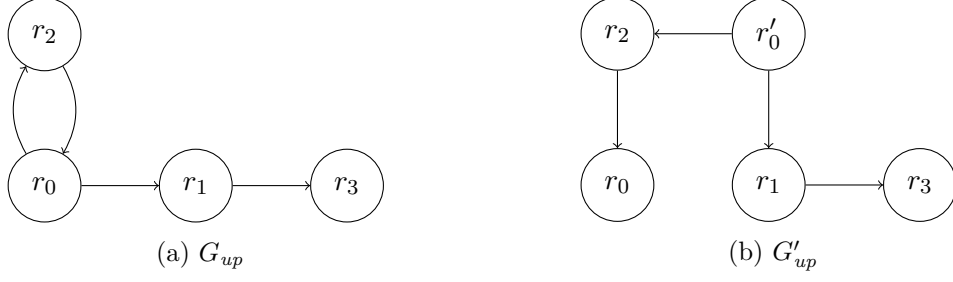


Figure 5.3: An example of a register dependency graph before (G_{up}) and after (G'_{up}) it was modified to be acyclic

Algorithm 8: Cycle removal from an update dependency graph

Input : An update dependency graph $G = (V, E)$

Output: G modified to be acyclic, and a bijection **node_map** mapping nodes to their new copies

```

1  $sccs \leftarrow SCCs(G)$ ;
2 while  $|sccs| < |V|$  do
3    $to\_split \leftarrow \{max\_out\_deg(s) \mid s \in sccs \wedge |s| > 1\}$ ;
4   foreach  $n \in to\_split$  do
5      $n' \leftarrow new\_node\_id()$ ;
6      $node\_map(n) \leftarrow n'$ ;
7      $V \leftarrow V \cup \{n'\}$ ;
8      $E \leftarrow E \cup \{(n', \cdot) \mid (n, \cdot) \in E\}$ ;
9      $E \leftarrow E \setminus \{e \in E \mid e = (n, \cdot)\}$ ;
10   $sccs \leftarrow SCCs(G)$ ;
11 return  $G, node\_map$ ;

```

(Line 6), adding the new copies to the set of vertices (Line 7), and moving the outgoing edges of the original node to the new node (Lines 8 and 9). After updating the graph, its SCCs are recomputed (Line 10). This is repeated until no cycles are present in the graph (in the worst-case scenario, each node has to be split).

After the graph and bijection is created, the nodes of the graph are ordered using a topological ordering algorithm. Lastly, it is necessary to build the new updates using the original ones, since we removed self-loops and *in* updates while creating the graph. This is done by going through the nodes in the topological order and assigning them their registers' original updates if they were not copied. Nodes that *are* copies, also get assigned the original updates of their original registers. The remaining updates are of the form $r \leftarrow r'$ and do not need any adjusting. We detect which registers were copied and which are copies using the bijection returned by Algorithm 8.

The Python package `networkx` [31] was used for graph representation, basic manipulation, and for computing strongly connected components and the topological ordering.

Example 5.3. Consider the update $up = \{r_0 \mapsto \{r_0, r_1, r_2, in\}, r_1 \mapsto \{r_3\}, r_2 \mapsto \{r_0\}, r_3 \mapsto \emptyset\}$. The graph G_{up} representing the dependencies in up is shown in Figure 5.3a. The first step in ordering the updates is ensuring that G_{up} is acyclic. Following Algorithm 8, strongly connected components are constructed, yielding the components $\{r_0, r_2\}$, $\{r_1\}$, and $\{r_3\}$. Since $\{r_0, r_2\}$ contains more than one vertex, it must contain a cycle. To break cycles in

a strongly connected component, we select the vertex with the highest outgoing degree to be split into two updates. In this case this is the vertex r_0 . After splitting the update for r_0 we obtain the graph G'_{up} shown in Figure 5.3b. G'_{up} is acyclic, and can be used to order updates.

To generate the updates, we find a topological ordering of G'_{up} , for example, $r'_0 r_2 r_0 r_1 r_3$. The updates would then be performed in the following order:

$$\begin{aligned} r'_0 &\leftarrow \{r_0, r_1, r_2, in\}, \\ r_2 &\leftarrow \{r_0\}, \\ r_0 &\leftarrow r'_0, \\ r_1 &\leftarrow \{r_3\}, \\ r_3 &\leftarrow \emptyset. \end{aligned}$$

Update Code Generation. The ordered updates need to be transformed into correct instructions. We show which instructions are generated for an update of register r . For any updates of type $r \leftarrow r'$ *only* the instruction **SWAP** r r' is generated. If r does not appear on the right hand side of the update and the register is not marked as empty in the macrostate, the **CLEAR** r instruction is generated. Then, if there are any items other than r and in on the right-hand side of the update, the instruction **UPDATE** r ($up(r) \setminus \{r, in\}$) is generated. Lastly, if in is on the right-hand side of the update, the instruction **ADDIN** r is generated.

5.2.4 Final VM Code

After the entire code is generated, it is necessary to remove jump labels and replace references to them with addresses. In our case, the addresses are indices into the instruction array. This is done textually in two passes. During the first pass, each label is stored as a key in a dictionary with the value being the index of the next non-label instruction. This is done by counting non-label instructions while processing the code and when a label is encountered, it gets stored into the dictionary with the current value of the counter. In the second pass, references to labels are replaced with the corresponding index stored in the dictionary. It is also necessary to go through the memory and replace every reference to a label with the correct address there.

Example 5.4. Figure 5.4 shows the VM code generated for the REBR $.(*)_1.*\backslash 1.*$. Specifically, Figure 5.4b shows the code before labels are removed, and Figure 5.4c shows the final code to be interpreted by the VM. Line numbers are added to code without labels for clarity. Figure 5.4a shows the DRSA from which the code is generated. The memory is not included since the decode instructions perform no jumps in this case because every transition label is Σ .

5.3 Code Execution

The code is executed by the VM implemented in C++. As command line arguments, the VM receives the file containing the program, the file containing the memory, and the number of registers the VM should use. Note that this number of registers is actually twice the number of registers of the original DRSA since we need the extra registers during register

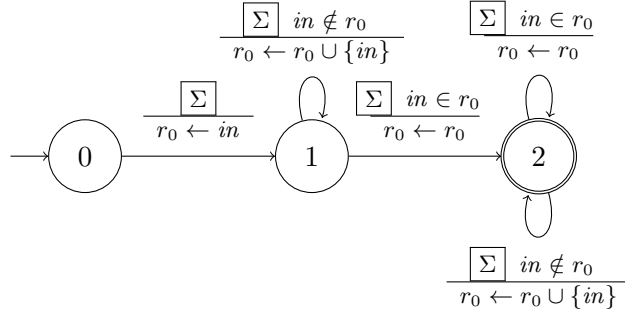
updates.² The registers are implemented as standard C++ unordered sets (hash tables). A simple parser was implemented to parse the program and create an array of instruction structures. The memory is passed as a json file. The nlohmann json library [28] was used to parse the memory file and create an array of decoding tree structures.

The execution loop itself is shown in Algorithm 9. The VM stores its state using the register array **R**, the instruction pointer *ip*, and the input value *in*. Inside the loop, the VM loads the instruction from *ip* (Line 5) and then switches based on the instruction's operation code (Line 6). First (and most complex) is the case of the **DECODE** operation code. The next UTF-8 character encoding is stored into *in*. A Boolean value indicating whether the operation succeeded (i.e., if there was a valid UTF-8 character to be returned) is stored into *succ* (Line 8). If *succ* is false, then the algorithm returns false (Line 9). Here we assume a valid UTF-8 input, otherwise an error should be returned if the input is an invalid UTF-8 encoding. Note that the character is stored into *in* as its UTF-8 encoding, not the Unicode codepoint. The UTF-8 encoding *in* is run through the decoding tree of the corresponding state (Line 10). The *label* field of the instruction is used to store the state's index into the memory. The variable *label* is checked for any special return values. This can either mean that there is no transition that can be taken using the symbol (Line 11) or that no jump should be performed (Line 12). Otherwise, *ip* is set to *label* (Line 15).

In the case of the **ACCEPT** operation code (Line 16), if the remaining input is empty, then true is returned (Line 17). Otherwise *ip* is incremented (Line 18). The operation code **FAIL** (Line 19) is handled by simply returning false (Line 20). In the case of **JUMP** (Line 21), *ip* is set to the instruction's label field (Line 22). For the **TEST** operation code (Line 23), the register from the instruction is first tested for membership of *in* (Line 24). If *in* is stored in the specified register, then *ip* is set to the label from the instruction (Line 25). Otherwise, no jump is performed (Line 27).

The **UPDATE** instruction (Line 28) is performed by adding all elements from all registers specified in the instruction's register list field to the register from the instruction's register field (Line 30). The **CLEAR** instruction (Line 32) clears the register from the instruction's register field (Line 33). In the case of the **SWAP** operation code (Line 35), the values of the two registers specified in the instruction are swapped (Line 36). The second register is stored as a singular element of the register list field. Lastly, the **ADDIN** instruction (Line 38) is handled by adding *in* to the register from the instruction (Line 39).

²During code generation, it would be possible to track the highest number of auxiliary registers in an update. The VM would then initialise exactly the number of registers needed to run the code. However, impact on performance would be negligible, as the extra registers are only initialized.



(a) DRSA constructed for the REBR .*(1.)1.*\1.*

```

state_0:
DECODE 0
0_0:
0_0_0:
ADDIN 0
JUMP state_1
state_1:
DECODE 1
1_0:
1_0_node_0:
TEST 0 1_0_0
JUMP 1_0_1
1_0_0:
JUMP state_2
1_0_1:
ADDIN 0
JUMP state_1
state_2:
ACCEPT
DECODE 2
2_0:
2_0_node_0:
TEST 0 2_0_0
JUMP 2_0_1
2_0_0:
JUMP state_2
2_0_1:
ADDIN 0
JUMP state_2

```

```

0: DECODE 0
1: ADDIN 0
2: JUMP 3
3: DECODE 1
4: TEST 0 6
5: JUMP 7
6: JUMP 9
7: ADDIN 0
8: JUMP 3
9: ACCEPT
10: DECODE 2
11: TEST 0 13
12: JUMP 14
13: JUMP 9
14: ADDIN 0
15: JUMP 9

```

(b) VM code with labels

(c) VM code after label processing

Figure 5.4: DRSA and VM code for the REBR .*(1.)1.*\1.*

Algorithm 9: Execution of code by the VM

Input : The arrays **prg** and **mem** storing the program and its memory, the number of registers **n**, and the input string **input**

Output: Boolean *true* if **input** is accepted and *false* otherwise

```
1 R  $\leftarrow$  init_regs(n);
2 ip  $\leftarrow$  0;
3 in  $\leftarrow$  0;
4 while true do
5   instr  $\leftarrow$  prg[ip];
6   switch instr.opcode :
7     case DECODE :
8       in, succ  $\leftarrow$  get_next_utf8_char(input);
9       if  $\neg$ succ then return false;
10      label  $\leftarrow$  traverse_decoding_tree(mem[instr.label], in);
11      if label = -2 then return false;
12      if label = -1 then
13        | ip  $\leftarrow$  ip + 1
14      else
15        | ip  $\leftarrow$  label
16      case ACCEPT :
17        if input.at_end() then return true;
18        ip  $\leftarrow$  ip + 1;
19      case FAIL :
20        return false;
21      case JUMP :
22        ip  $\leftarrow$  instr.label;
23      case TEST :
24        if in  $\in$  R[instr.reg] then
25          | ip  $\leftarrow$  instr.label;
26        else
27          | ip  $\leftarrow$  ip + 1;
28      case UPDATE :
29        foreach r  $\in$  instr.reg_list do
30          | R[instr.reg].insert_from(r);
31        ip  $\leftarrow$  ip + 1;
32      case CLEAR :
33        R[instr.reg].clear();
34        ip  $\leftarrow$  ip + 1;
35      case SWAP :
36        swap(R[instr.reg], R[instr.reg_list[0]]);
37        ip  $\leftarrow$  ip + 1;
38      case ADDIN :
39        R[instr.reg].insert(in);
40        ip  $\leftarrow$  ip + 1;
```

Chapter 6

Experiments

This chapter covers experiments made on the improved prototype and the VM matcher implemented during this work. First, they are compared against the old prototype and each other, then they are compared against other state-of-the-art regex matchers.

6.1 Experimental Setup

The experiments evaluate the performance of matchers on ReDoS attack vectors involving back-references. The scenario for our evaluation is one with a fixed compiled regex and a malicious string (attack string) supplied by the attacker. The experiments were run on a Ubuntu GNU/Linux machine with the AMD EPYC 9124 CPU with 16 cores at 3.0 GHz and 94 GiB of memory.

Regexes and Regex Matchers. As our regexes, we started with 3,252 real-world regexes with back-references obtained from the comprehensive data set *Lingua Franca* [13] and 8,794 regexes from the benchmarks of the ReDoS generator RENGAR [43], for a total of 12,046 regexes. From these, we picked regexes with single-letter back-references, which gave us 3,299 regexes (27 %). By single-letter, we mean regexes that have no back-referenced capture groups of length over one.

We denote the current version of the prototype `rsaregex`, the version of the prototype before the improvements from this work `oldrsa`, and the C++ VM matcher `rsacpp`. We compared against the state-of-the-art tools `grep` [24], `pcre2` [23], Python RE [29] (denoted as `re`), standard library matchers of JavaScript [18], Java [34], and .NET [30] (denoted as `js`, `java`, and `.net` respectively). All non-RSA based matchers mainly use backtracking when confronted with back-references.

ReDoS Inputs. In general, there are several flavours of ReDoS considered in the literature. In our evaluation, we consider the scenario where there is a regex (known to the attacker) running on a server (or a network intrusion detection system, a packet filter, etc.) and the attacker provides an input that exhibits a costly matching, which is, to the best of our knowledge, the most realistic setting, since it has already led to several real-world ReDoSes [37, 2]¹. The input of every experiment is a pair of a regex and a (potentially) mali-

¹Another setting considered in the literature is, e.g., when the attacker can provide both the regex and the input. While such a situation may occur in the real-world, it is less common and much easier to exploit when back-references are enabled due to the NP-hardness of their matching [1].

Table 6.1: Compilation statistics for **rsaregex**, **rsacpp**, and **oldrsa**, time is in seconds

tool	mean	median	std. dev.	TOs	errors
rsaregex	0.58	0.14	2.09	21	4
rsacpp	1.16	0.32	2.80	21	4
oldrsa	7.90	2.46	14.57	109	598

Table 6.2: Matching time statistics for **rsaregex**, **rsacpp**, and **oldrsa**, in seconds

tool	mean	median	std. dev.
rsaregex	0.10	0.066	0.096
rsacpp	0.0017	0.0014	0.0015
oldrsa	0.89	0.56	1.20

cious input string, which was created using the ReDoS generator RENGAR [43]. RENGAR was able to provide an attack vector for 1,330 regexes from the data set (40%). RENGAR generated strings of length between 55 and 90,124 characters, with an average of 34,521 characters. The timeout limit was set to 100 s.

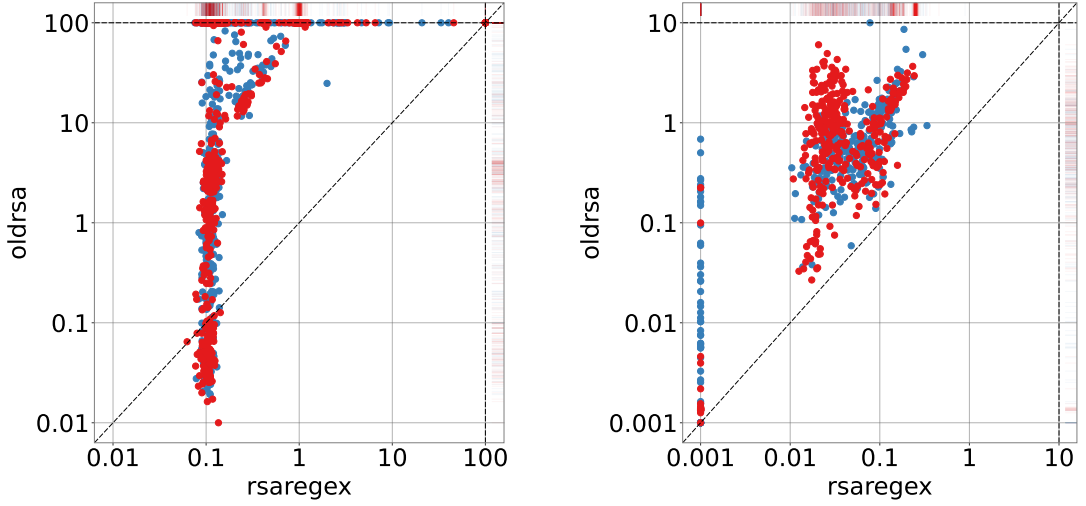
6.2 Evaluating Improvements to RSA-based Matching

This section evaluates the improvements made to RSA-based matching implemented in this work, which includes both improvements of the prototype and the VM matcher. We evaluate in terms of matching speed, compilation speed, and the number of compiled regexes. For **rsaregex** and **oldrsa**, compilation includes everything from the start of parsing to the end of determinisation. For **rsacpp**, it additionally includes code generation, code parsing, and loading the generated memory.

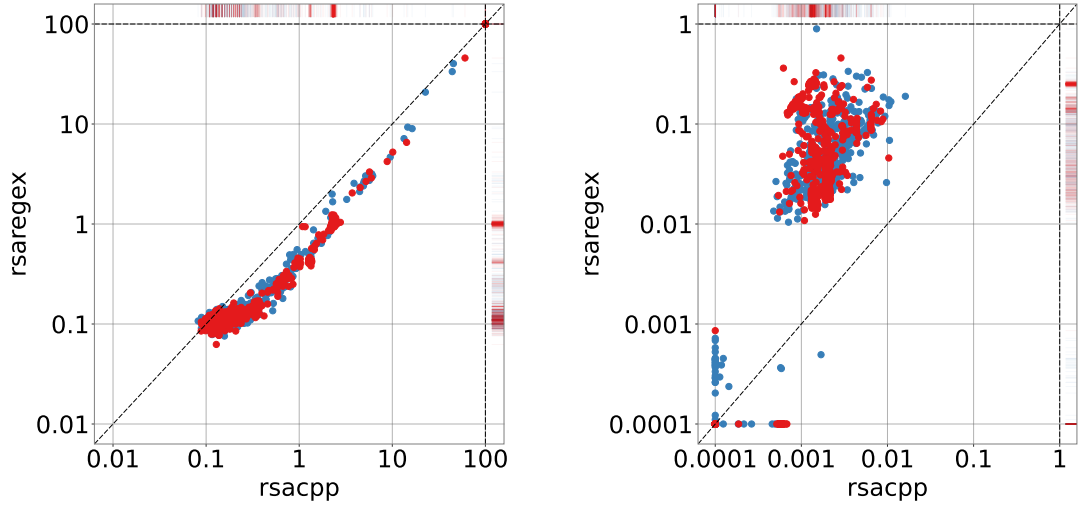
Prototype Improvements. We compare the prototype’s current version, **rsaregex** to its old version, **oldrsa**. Table 6.1 shows compilation statistics, which show that **rsaregex** is substantially faster and more stable, as evidenced by the lower mean, median and standard deviation values; **rsaregex** also experiences a lower number of timeouts. Significantly more regexes are also supported by **rsaregex** than **oldrsa**. All 4 **rsaregex** errors are caused by overapproximation during determinisation. For **oldrsa**, 592 errors are due to capture groups of length 0–1 and the remaining 6 errors are due to overapproximation. Figure 6.1a shows the compilation times of **rsaregex** and **oldrsa** in a scatter plot, which shows that **rsaregex** is only slower than **oldrsa** in some small cases (specifically cases that take up to about 0.1 s).

Table 6.2 shows that **rsaregex** is also much faster and more stable than **oldrsa** in terms of matching speed. Figure 6.1b shows that **rsaregex** is indeed faster than **oldrsa** in all cases when it comes to matching.

VM Matcher. We compare the VM matcher **rsacpp** to the **rsaregex**. Table 6.1 shows that compilation time for **rsacpp** is about twice as long as for **rsaregex** on average. The median compilation time of **rsacpp** is slightly more than double that of **rsaregex**. The standard deviation is also slightly higher for **rsacpp** than **rsaregex**. This slowdown is mostly due to the code generation that is performed after determinisation. However, no



(a) Compilation times of **rsaregex** against **oldrsa** (b) Matching times of **rsaregex** against **oldrsa**



(c) Compilation times of **rsacpp** against **rsaregex** (d) Matching times of **rsacpp** against **rsaregex**

Figure 6.1: Comparison of runtimes, axes are logarithmic, colours distinguish datasets: ● Lingua Franca, ● RENGAR.

additional timeouts are present for **rsacpp**. Figure 6.1c shows the scatter plot comparing the two, which shows the expected pattern of slowdown caused by the code generation.

Concerning matching speed, Table 6.2 shows that, on average, **rsacpp** is approximately $60\times$ faster than **rsaregex**. Both the median and the standard deviation of **rsacpp** are also around two orders of magnitude smaller than those of **rsaregex**. Figure 6.1d shows the scatter plot of matching times of the two tools. It shows that **rsacpp** is significantly faster for all runs, except some of the very fast ones (< 0.002), which are sensitive to noise during measurements.

6.3 Comparison with Other Matchers

To compare against other matchers, we selected the 1,305 regexes that compiled on **rsaregex** (or, equivalently, on **rsacpp**) within the time limit without errors. In Table 6.3, we show

Table 6.3: A frequency table of the run times and statistics for **rsacpp**, **rsaregex**, and the other matchers on the 1,305 supported regexes with the RENGAR-generated inputs. The column **TOs** denotes timeout (> 100 s), column > 1 s (> 0.1 s) is the sum of the values in all columns over 1 s (0.1 s), and the **errors** column shows the numbers of unsupported regexes. The numbers in the right-hand part of the table are for successful runs of the given tool only and are in seconds.

tool	< 0.1 s	0.1 – 1 s	1 – 5 s	5 – 10 s	10 – 50 s	50 – 100 s	TOs	> 0.1 s	> 1 s	errors	mean	std. dev
rsacpp	1305	0	0	0	0	0	0	0	0	–	0.002	0.002
rsaregex	753	552	0	0	0	0	0	552	0	–	0.105	0.096
oldrsa	92	344	183	3	1	0	0	626	282	682	0.889	1.205
grep	1153	19	6	1	16	7	17	66	47	86	0.846	7.102
re	1084	106	40	14	10	3	48	221	115	0	0.511	4.293
pcre2	1031	27	15	4	4	0	2	52	25	222	0.14	1.179
js	645	59	8	1	12	1	37	118	59	542	0.539	3.711
java	554	76	18	5	7	5	23	134	58	617	0.975	6.491
.net	653	41	30	1	11	3	28	114	73	538	0.852	6.009

a frequency table (the table form of a histogram) of run times of **rsacpp**, **rsaregex**, and the other matchers on the 1,305 supported input regexes. We note that some of the regex matchers failed on a considerable number of the input regexes—this is due to different dialects of regexes used by the matchers and should not be considered their failure (**rsaregex** uses Python’s dialect, which is why **re** has 0 errors). The use of different dialects means that the semantics of one regex differs based on the matcher [13]; while this would be an issue if we were cross-comparing the performance of the matchers, in this experiment, we are focusing on ReDoS vulnerability, where the concrete semantics of a regex is not so important.

The results show that both **rsacpp** and **rsaregex**’s run time for all benchmarks is below one second, and **rsacpp**’s run time even stays below 0.1 seconds for all benchmarks. In contrast, the other matchers struggle on the attack vectors considerably, taking tens of seconds or even exceeding the 100 s timeout. Considering that the normal matching time on such strings is counted in milliseconds, they can indeed be considered manifestations of vulnerabilities. The columns > 1 s (> 0.1 s) can be seen as counting the number of successful ReDoS attacks if a ReDoS threshold is set to 1 s (0.1 s). Here, after **rsacpp** (and **rsaregex** in the case of the 1 s threshold), the best-performing matchers are **pcre2** and **grep**. Note that **pcre2** has an internal timeout limit, which was increased to the maximum for a better comparison to the other matchers. In 17 cases, **pcre2** reached the internal timeout anyway; these cases are included in the error column.

In the right-hand part of the table, we give statistics about the run times of the matchers on the instances where they did not time out (or fail): the arithmetic mean and the standard deviation. The median is left out since the numbers were small and could contain imprecisions. However, the numbers suggest that in the median case (i.e., a case without a vulnerability) the backtracking matchers are slightly faster than **rsacpp**. In the other two metrics, the best performing matcher is clearly **rsacpp**, with the lowest mean and standard deviation, both by multiple orders of magnitude.

In summary, all backtracking matchers performed poorly on these attacks, with all of them having multiple instances of reaching the 100 s timeout. Our matcher, **rsacpp**, proved to be the only matcher immune to the generated ReDoS attacks.

Chapter 7

Conclusion

We have implemented a virtual machine regex matcher **rsacpp** based on register set automata (RSA). The matcher significantly improves over a previous prototype. It works by converting a regex into a deterministic RSA and generating code; the code is then executed for each given input. Once compiled, **rsacpp** can efficiently match regexes with backreferences to single-letter capture groups (i.e., capture groups of length at most one).

We have experimentally evaluated the performance of **rsacpp** under simulated ReDoS attacks and compared it against state-of-the-art regex matchers and the previous prototype. The simulated scenario is one with a fixed regex, and an adversarial input created by an attacker. The regexes in our experiments are regexes from real world projects. To create the attack inputs, we used the ReDoS generator RENGAR [43], which generated malicious inputs to over 1,300 regexes.

The results show that while all of the other matchers considerably struggled under the simulated ReDoS attacks, **rsacpp** was immune to all of them. Every other matcher had multiple instances of reaching the 100s timeout. In contrast, **rsacpp** stayed under 0.1s for every attack. Additionally, **rsacpp** exhibited the lowest arithmetic mean and standard deviation of matching times (even when excluding timeouts and errors of each respective). The experimental evaluation and implementation from this thesis were part of a publication accepted to PLDI'26 [22].

Discussion. Even though our matcher proved to be the only one immune to ReDoS attacks, there are limitations to our approach. Not all regexes with back-references are supported. This is not surprising, considering that their matching is NP-complete in general. However, some regexes are not supported even in the case of back-references limited to single-letter capture groups.

Next, the approach is only suitable for applications where a large compile time of the regex does not cause issues. In other words, it is not suitable for matching regexes created at runtime since regex compilation is slow. Allowing users to supply regexes would constitute a ReDoS threat.

Lastly, there could possibly be attack vectors our matcher is vulnerable to, which might be different in kind to what backtracking matchers are vulnerable to. Currently, we are not aware of such attacks, but more research is required on this topic.

Future Work. As future work, we would like to address the concerns raised in the discussion. There are ideas to increase the class of supported regexes with back-references to

longer constant-length (i.e., the capture group matches words of the same length across all possible runs) capture groups.

To make RSA-based matching viable for regexes created at runtime, an *on-the-fly* algorithm could possibly be implemented. This would mean creating the RSA lazily during matching, which would probably require scrapping the current prototype and re-implementing the entire matcher with efficiency in mind. As already mentioned above, more research on what inputs the RSA-based approach might be vulnerable is necessary.

Bibliography

- [1] AHO, A. V. Algorithms for Finding Patterns in Strings. In: LEEUWEN, J. van, ed. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990, p. 255–300.
- [2] BALDWIN, A. *Regular Expression Denial of Service affecting Express.js* [online]. 2016. Available at: <https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43>. [cit. 2026-02-03].
- [3] BANERJEE, A.; CHATTERJEE, K. and GUHA, S. Set Augmented Finite Automata over Infinite Alphabets. *CoRR*, 2023, abs/2311.06514. Available at: <https://doi.org/10.48550/arXiv.2311.06514>.
- [4] BARRIÈRE, A. and PIT-CLAUDEL, C. Linear Matching of JavaScript Regular Expressions. *Proc. ACM Program. Lang.*, 2024, vol. 8, PLDI, p. 1336–1360. Available at: <https://doi.org/10.1145/3656431>.
- [5] BERGLUND, M. and MERWE, B. van der. Re-examining regular expressions with backreferences. *Theor. Comput. Sci.*, 2023, vol. 940, Part, p. 66–80. Available at: <https://doi.org/10.1016/j.tcs.2022.10.041>.
- [6] BJÖRKLUND, H. and SCHWENTICK, T. On notions of regularity for data languages. *Theor. Comput. Sci.*, 2010, vol. 411, 4-5, p. 702–715. Available at: <https://doi.org/10.1016/j.tcs.2009.10.009>.
- [7] BOJAŃCZYK, M. *Lecture notes for Advanced Topics in Automata (Języki, Automaty i Obliczenia 2): 8. Automata with infinite alphabets* [online]. 2015/2016. Available at: <https://www.mimuw.edu.pl/~bojan/20152016-2/języki-automaty-i-obliczenia-2/8-automata-with-infinite-alphabets>. [cit. 2026-01-26].
- [8] BOJANCZYK, M.; MUSCHOLL, A.; SCHWENTICK, T.; SEGOUFIN, L. and DAVID, C. Two-Variable Logic on Words with Data. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, p. 7–16. Available at: <https://doi.org/10.1109/LICS.2006.51>.
- [9] BRIGGS, P. and TORCZON, L. An Efficient Representation for Sparse Sets. *LOPLAS*, 1993, vol. 2, 1-4, p. 59–69. Available at: <https://doi.org/10.1145/176454.176484>.
- [10] BRYANT, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 1986, vol. 35, no. 8, p. 677–691. Available at: <https://doi.org/10.1109/TC.1986.1676819>.

- [11] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. and STEIN, C. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8.
- [12] COX, R. *Regular Expression Matching: the Virtual Machine Approach* [online]. 2009. Available at: <https://swtch.com/~rsc/regexp/regexp2.html>. [cit. 2026-05-14].
- [13] DAVIS, J. C.; IV, L. G. M.; COGHLIN, C. A.; SERVANT, F. and LEE, D. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In: DUMAS, M.; PFAHL, D.; APEL, S. and RUSSO, A., ed. *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, p. 443–454. Available at: <https://doi.org/10.1145/3338906.3338909>.
- [14] DEMRI, S. and LAZIC, R. LTL with the Freeze Quantifier and Register Automata. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, p. 17–26. Available at: <https://doi.org/10.1109/LICS.2006.31>.
- [15] FREE SOFTWARE FOUNDATION. *GNU grep 3.11 manual* [online]. 2024. Available at: <https://www.gnu.org/software/grep/manual/grep.html>. [cit. 2024-11-17].
- [16] FREYDENBERGER, D. D. and SCHMID, M. L. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 2019, vol. 105, p. 1–39. Available at: <https://doi.org/10.1016/j.jcss.2019.04.001>.
- [17] GOOGLE. *RE2 Issue Tracker: Feature request #101: Add backreference support* [online]. 2016. Available at: <https://github.com/google/re2/issues/101>. [cit. 2026-02-03].
- [18] GOOGLE. *V8 regular expression source code* [online]. 2024. Available at: <https://github.com/v8/v8/tree/main/src/regexp>. [cit. 2024-11-17].
- [19] GOOGLE. *RE2* [online]. 2025. Available at: <https://github.com/google/re2>. [cit. 2025-03-25].
- [20] GRIGORE, R. and TZEVELEKOS, N. History-Register Automata. *Log. Methods Comput. Sci.*, 2016, vol. 12, no. 1. Available at: [https://doi.org/10.2168/LMCS-12\(1:7\)2016](https://doi.org/10.2168/LMCS-12(1:7)2016).
- [21] GROSS, B.; UTZ, D.; ASCHERL, S.; VOLLMEIER, M.; WANSCHURA, K. et al. *OpenMoji 17.0.0* [online]. 2026. Available at: <https://openmoji.org>. [cit. 2026-05-14].
- [22] HAVLENA, V.; HOLÍK, L.; LENGÁL, O.; VAŠÁK, J. and GULČÍKOVÁ, S. *Towards Efficient Matching of Regexes with Backreferences using Register Set Automata (Technical Report)*. 2026. Available at: <https://arxiv.org/abs/2205.12114>. Accepted to PLDI'26.
- [23] HAZEL, P. *Perl-compatible Regular Expressions* [online]. 10.43. February 2024. Available at: <https://www.pcre.org/>. [cit. 2024-11-17].
- [24] HEARTEL, M. *GNU grep* [online]. 3.7. September 2022. Available at: <https://www.gnu.org/software/grep/>. [cit. 2024-11-17].

- [25] INTEL. *HyperScan manual, unsupported features* [online]. 2018. Available at: <https://intel.github.io/hyperscan/dev-reference/compilation.html#unsupported-constructs>. [cit. 2026-02-03].
- [26] KAMINSKI, M. and FRANCEZ, N. Finite-Memory Automata. *Theor. Comput. Sci.*, 1994, vol. 134, no. 2, p. 329–363. Available at: [https://doi.org/10.1016/0304-3975\(94\)90242-9](https://doi.org/10.1016/0304-3975(94)90242-9).
- [27] KARP, R. M. Reducibility Among Combinatorial Problems. In: MILLER, R. E. and THATCHER, J. W., ed. *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Plenum Press, New York, 1972, p. 85–103. The IBM Research Symposia Series. Available at: https://doi.org/10.1007/978-1-4684-2001-2_9.
- [28] LOHMANN, N. *JSON for Modern C++* [online]. 3.12.0. 2026. Available at: <https://json.nlohmann.me/>. [cit. 2026-05-14].
- [29] LUNDH, F. and KUHLING, A. M. *Python Standard Library: re module* [online]. 3.10.12. June 2023. Available at: <https://docs.python.org/3/library/re.html>. [cit. 2024-11-17].
- [30] MICROSOFT. *.NET 8.0.110 regular expressions in System.Text.RegularExpressions* [online]. 2024. Available at: <https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions?view=net-8.0>. [cit. 2024-11-17].
- [31] NETWORKX DEVELOPERS. *NetworkX* [online]. 3.4.2. 2025. Available at: <https://networkx.org>. [cit. 2026-05-14].
- [32] 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. Available at: <https://doi.org/10.1145/1013560.1013562>.
- [33] NOGAMI, T. and TERAUCHI, T. Efficient Matching of Some Fundamental Regular Expressions with Backreferences. In: GAWRYCHOWSKI, P.; MAZOWIECKI, F. and SKRZYPCZAK, M., ed. *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025, Warsaw, Poland, August 25-29, 2025*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, vol. 345, p. 81:1–81:19. LIPIcs. Available at: <https://doi.org/10.4230/LIPIcs.MFCS.2025.81>.
- [34] OPENJDK. *JDK 17 API Documentation: Regular Expressions* [online]. 2024. Available at: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/package-summary.html>. [cit. 2024-11-17].
- [35] SCHMID, M. L. Regular Expressions with Backreferences: Polynomial-Time Matching Techniques. *J. Autom. Lang. Comb.*, 2024, vol. 29, 2-4, p. 321–357. Available at: <https://doi.org/10.25596/jalc-2024-321>.
- [36] SCHMITZ, S. and SCHNOEBELEN, P. *Algorithmic Aspects of WQO Theory* [online]. Lecture Notes. France, 2012. Available at: <https://cel.hal.science/cel-00727025>. [cit. 2026-01-27].

- [37] STACK EXCHANGE. *Outage Postmortem* [online]. 2016. Available at: <https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016>. [cit. 2026-02-03].
- [38] THE UNICODE CONSORTIUM. *The Unicode Standard, Version 17.0* [online]. 2025. Available at: <https://www.unicode.org/versions/Unicode17.0.0/core-spec/>. [cit. 2026-05-14].
- [39] THOMPSON, K. Regular Expression Search Algorithm. *Commun. ACM*, 1968, vol. 11, no. 6, p. 419–422. Available at: <https://doi.org/10.1145/363347.363387>.
- [40] VAŠÁK, J. *Formal Models for Data Languages* [online]. Brno, 2024. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor LENGÁL, O. Available at: <https://www.vut.cz/studenti/zav-prace/detail/154542>. [cit. 2026-01-27].
- [41] W3TECHS. *Usage statistics of UTF-8 for websites* [online]. 2026. Available at: <https://w3techs.com/technologies/details/en-utf8>. [cit. 2026-05-14].
- [42] WANG, X.; HONG, Y.; CHANG, H.; PARK, K.; LANGDALE, G. et al. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In: LORCH, J. R. and YU, M., ed. *NSDI’19*. USENIX Association, 2019, p. 631–648. Available at: <https://www.usenix.org/conference/nsdi19/presentation/wang-xiang>.
- [43] WANG, X.; ZHANG, C.; LI, Y.; XU, Z.; HUANG, S. et al. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, p. 2427–2443.
- [44] YANG, L.; GANAPATHY, V.; MANADHATA, P. K. and WU, Y. A novel algorithm for pattern matching with back references. In: *34th IEEE International Performance Computing and Communications Conference, IPCCC 2015, Nanjing, China, December 14-16, 2015*. IEEE Computer Society, 2015, p. 1–8. Available at: <https://doi.org/10.1109/PCCC.2015.7410264>.

Appendix A

Contents of the Included Storage Media

/	
└─ src/.....	Implementation source files
└─ rsaregex/.....	Python module implementing rsaregex
└─ vmmatcher/.....	Directory containing rsacpp source files
└─ README.md	Usage guide
└─ rsa-matcher.py.....	Implementation of a grep-like program using rsaregex
└─ tex-src/.....	L ^A T _E X source files of this thesis
└─ thesis.pdf.....	This thesis in pdf version

Appendix B

PLDI Publication

This appendix (starting on the following page) contains the publication created during this thesis that was accepted to PLDI 2026 [\[22\]](#).

Towards Efficient Matching of Regexes with Backreferences using Register Set Automata

VOJTĚCH HAVLENA, Brno University of Technology, Czech Republic

LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic and Aalborg University, Denmark

ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic

JAN VAŠÁK, Brno University of Technology, Czech Republic

SABÍNA GULČÍKOVÁ, Brno University of Technology, Czech Republic

Matching regexes (regular expressions) is a common problem in many areas of computer science, with requirements on high speed and robust performance. Regexes with backreferences allow one to express certain patterns (even beyond regular) concisely, however, since the matching is usually done by backtracking, the matching speed can degrade to a degree that constitutes a service failure or a security threat. To facilitate high-speed matching of such regexes, we propose *register set automata* (RSAs), an extension of register automata where registers can contain *sets* of symbols (from a potentially infinite alphabet) and the following operations are supported: adding input values to registers, merging or clearing registers, and testing whether a register contains a value. We show that a large class of *register automata* can be transformed into *deterministic* RSAs, which can serve as a basis for fast matching of a family of regexes with single-letter capture groups and backreferences. We also give a derivative-based algorithm for transforming a large class of regexes with backreferences to register automata and show that the time complexity of matching is linear and quadratic to the length of the input for finite and infinite alphabets respectively. Our prototype implementation of a regex matcher shows that our approach can significantly improve the robustness of state-of-the-art regex matchers on regexes with backreferences. We also study the theoretical properties of the model and show that the emptiness problem for RSAs is decidable and complete for the F_ω class and that RSAs are incomparable in expressive power to other popular automata models over data words.

CCS Concepts: • **Security and privacy** → **Denial-of-service attacks**; • **Theory of computation** → **Automata over infinite objects**; **Regular languages**; • **Applied computing** → *Document searching*.

Additional Key Words and Phrases: regular expression matching, backreferences, ReDoS, determinization, Antimirov's derivatives, register automata, register set automata

ACM Reference Format:

Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Jan Vašák, and Sabína Gulčíková. 2026. Towards Efficient Matching of Regexes with Backreferences using Register Set Automata. *Proc. ACM Program. Lang.* 10, PLDI, Article 203 (June 2026), 26 pages. <https://doi.org/10.1145/3808281>

1 Introduction

Regular expression (regex) matching is a task performed routinely in computer programs, such as in searching, data validation, parsing, finding and replacing, data leak detection, or syntax highlighting. Studies show that 30–40 % of Java, JavaScript, and Python software use regex matching [15]. In

Authors' Contact Information: [Vojtěch Havlena](mailto:ihavlena@fit.vutbr.cz), Brno University of Technology, Brno, Czech Republic, ihavlena@fit.vutbr.cz; [Lukáš Holík](mailto:holik@fit.vutbr.cz), Brno University of Technology, Brno, Czech Republic and Aalborg University, Aalborg, Denmark, holik@fit.vutbr.cz; [Ondřej Lengál](mailto:lengal@fit.vutbr.cz), Brno University of Technology, Brno, Czech Republic, lengal@fit.vutbr.cz; [Jan Vašák](mailto:xvasak01@stud.fit.vutbr.cz), Brno University of Technology, Brno, Czech Republic, xvasak01@stud.fit.vutbr.cz; [Sabína Gulčíková](mailto:sgulci00@stud.fit.vutbr.cz), Brno University of Technology, Brno, Czech Republic, sgulci00@stud.fit.vutbr.cz.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART203

<https://doi.org/10.1145/3808281>

Proc. ACM Program. Lang., Vol. 10, No. PLDI, Article 203. Publication date: June 2026.

many of these applications, performance of regex matching is crucial, such as in various online services, where underperforming regex matching over user input can cause the server to become unresponsive and unusable. If such a situation is caused intentionally by a malicious user, we talk about the so-called *regular expression denial of service* (ReDoS) attack [1]. ReDoS is a real-world threat; it caused, for instance, the 2016 outage of StackOverflow [22] or rendered vulnerable websites that were using the Express.js framework [4].

While there are matchers that have a reasonably robust performance on basic regexes—such as `grep` [21], `RE2` [28], or `HyperScan` [73]—their performance on *extended regexes* can quickly deteriorate [68] (or they do not support the particular extensions at all [27, 34]). Examples of such extensions are, e.g., bounded counters [68, 69], lookaheads and lookbehinds, or capture groups and backreferences, which are the topic of this paper. The performance degrades mainly due to the inability of matchers to use the fastest automata-based algorithms that have constant or at most linear per-character cost. These algorithms are based on the automata determinization,¹ which cannot easily accommodate the syntactic features of extended regexes (the language of a regex with backreferences may even not be regular). The matchers then usually fall back to *backtracking*, which is much slower on average and exponential at worst.

Pattern matching using regular expressions with backreferences is performed ubiquitously, e.g., in validating user inputs on web pages, processing text using the `grep` and `sed` tools, transforming XML documents, or detecting network incidents [8, 54]. Consider, for instance, the (extended) regex

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

where we use the wildcard "." to denote any symbol except the semicolon ";" (i.e., it stands for the character class "[^;]"), parentheses "(...)" to denote the so-called *capture groups*, and "\x" to denote a *backreference* to the string captured by the x'th capture group (the semicolon ";" serves in the regexes as a delimiter). Intuitively, the regex matches input strings w that can be seen as a concatenation of three strings $w = uvz$ such that u has the structure $u = u_1;u_2;u_3$ with $u_1, u_2, u_3 \in (\Sigma \setminus \{;\})^+$, v is a string of three characters $a_3a_2a_1$ such that $a_i \in u_i$ for $i \in \{1, 2, 3\}$, and $z \in (\Sigma \setminus \{;\})^*$ (such a regex can describe, e.g., a simplified version of the match rule of some XML transformation). Trying (unsuccessfully) to find a match of the regex in the randomly generated 42-character-long string

"ah;jk2367ash;1a5akv451wkjb9f.dj5fqkbsfyrf"

using the state-of-the-art PCRE2 regex matcher available at regex101.com [53] takes 10,374 steps before reporting *no match*.² Ideally, the matcher should take only 42 steps, one step for every character in the string. This $\frac{10,374}{42} = 247\times$ 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. From a theoretical point of view, this inadequate performance is hardly a surprise, since matching of regexes with backreferences is an NP-hard problem [2]. This theoretical obstacle, however, does not need to be a show-stopper, as for many regexes appearing in practice, there is still hope that a matching algorithm with a time complexity linear (or at most quadratic) to the size of the input is possible.

As mentioned above, practical means of avoiding backtracking in such cases are not available since the automata determinization does not support backreferences. Indeed, neither of the two currently most advanced regex matchers in the industry, `RE2` [28] and `HyperScan` [73], supports

¹In practice, the matchers are based on Thompson's algorithm [66], which does not build the (possibly prohibitively large) DFA *a priori* but, instead, *on the fly*, while using cache to store already constructed parts of the DFA.

²Finding a match of the regex $/(\.)*;.*(\.)*;.*(\.)*\setminus 3 \setminus 2 \setminus 1/$ in the same text took 169,379 steps before *no match* was reported. This regex is more challenging than $R_{\setminus 3 \setminus 2 \setminus 1}$ —it does not use delimiters (semicolons in $R_{\setminus 3 \setminus 2 \setminus 1}$).

backreferences, due to a missing efficient algorithm [27, 34]. The main obstacle is a lack of a suitable deterministic automata model with a fast membership test that would support backreferences.

In this paper, we develop such a formal model, particularly suited for fast matching of a class of these regexes where capture groups are single-letter (the single-letter backreferences constitute a large portion of backreferences in our dataset collected for real-world applications). The formal model are *register set automata* (RSAs), an extension of *register automata* [18, 37] (RAs) where registers can contain sets of symbols instead of just single symbols (as for RAs). Deterministic RSAs (DRSAs) can be simulated in an efficient and robust regex-matching algorithm in time linear to the length of the input text, for finite alphabets, or quadratic to the size of the input for infinite alphabets. Our key result is a *RA to DRSA determinization (semi-)algorithm*; we also propose a partial-derivative-based algorithm for compiling a regex with single-letter backreferences to an RA that completes the matching workflow. We implemented a prototype matcher based on these results and experimented with a sample of regexes with single-letter backreferences extracted from a comprehensive real-world benchmark set. Although our determinisation is not complete (it may fail), the experiments show that it is quite reliable, and our matcher indeed significantly improves predictability of matching by state-of-the-art matchers and reduces the danger of ReDoS.

Theory of Register Set Automata. We also deliver a number of positive theoretical results related to our new RSA automata model, which place it into the landscape of automata over infinite alphabets. First, the model strictly generalizes RAs [18, 37] and is incomparable to (one-way) alternating RAs, another popular powerful generalization of RAs [18]. Besides the aforementioned RA to RSA determinization, the core property of RSAs is that their emptiness problem is decidable, although for a higher price than for RAs—the complexity blows from PSPACE-complete for RAs to F_ω -complete (i.e., Ackermannian) for RSAs.

Importantly, since we can now determinize RAs to RSAs and test emptiness of RSAs, we can use them to test language inclusion of RAs by the standard approach: by determinizing, complementing, and testing emptiness of the intersection (complementing a DRSA is done easily by swapping final and non-final states). Testing RA inclusion is an essential problem in many of their applications, such as in their minimization, learning [10, 26, 36], checking for fixpoint in regular model checking [12], checking XML schema subsumption [67], verification of parameterized concurrent programs with shared memory [35], and is an essential component of the RA toolkit. Unfortunately, inclusion of RAs is in general an undecidable problem [48], which has forced researchers to either find ways to approximate the inclusion test—e.g., via several membership tests [26, 36] or by an abstraction refinement semi-algorithm using interpolation [35]—or restrict themselves to other models with decidable inclusion problem, such as deterministic RAs (whose expressive power is quite limited) [46] or session automata [10]. We note that inclusion of languages described by regexes with backreferences is also undecidable [24].

Contribution. Let us summarise the main contributions of the paper:

- (1) Introduction of the model of register set automata and its theoretical analysis: Showing its closure properties, placing the RSA model into the landscape of automata-with-registers models, and proving F_ω -completeness of the emptiness problem for RSAs by showing interreducibility with the coverability problem for transfer Petri nets, discussing the power of several extensions of the model.
- (2) Designing a (semi-)algorithm that can determinize an RA into a DRSA and showing that it is complete for the class of languages that can be obtained from RAs with one register and no disequality tests by Boolean operations (i.e., union, intersection, and complement).
- (3) Developing a partial-derivative based algorithm for converting a large class of regexes with backreferences to RAs.

- (4) Showing that DRSA-based regex matching can be done in linear or quadratic time, depending on whether the size of the alphabet is finite or infinite respectively.
- (5) Experimentally confirming that simulation of the deterministic RSA obtained by our algorithm is a practical matching algorithm for regexes with single-letter backreferences, and that it is significantly more resilient against ReDoS than state-of-the-art matchers.

2 Preliminaries

Sets and Functions. 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 $n \in \mathbb{N}$ to denote the set $\{1, \dots, n\}$ (we note that $[0] = \emptyset$). We sometimes use “.” to denote an *ellipsis*, i.e., a value that can be ignored. For a (partial) function $f: A \rightarrow B$, a set $C \subseteq B$, and an element $d \in B$, we use $f[C \mapsto d]$ to denote the function $f[C \mapsto d](x) = d$ if $f(x) \in C$ and $f[C \mapsto d](x) = f(x)$ otherwise. Furthermore, for a set $X \subseteq A$, we use $f|_X$ to denote the restriction of f to X defined as $f|_X = f \cap (X \times B)$.

Data Words. Let us fix a finite nonempty *alphabet* Σ and an infinite *data domain* $\mathbb{D}$. A (finite) *data word* of length n is a function $w: [n] \rightarrow (\Sigma \times \mathbb{D})$; we use $|w| = n$ to denote its length and $w_1, \dots, w_n$ to denote its symbols. The *empty word* of length 0 is denoted ϵ . We use $\Sigma[w]$ and $\mathbb{D}[w]$ to denote the *projection* of w onto the respective domain (e.g., if $w = \langle a, 1 \rangle \langle b, 2 \rangle \langle b, 3 \rangle$, then $\Sigma[w] = abc$ and $\mathbb{D}[w] = 123$) and, given $a \in \Sigma$, we use $a[w_i]$ as a shortcut for $\Sigma[w_i] = a$.

Register Automata on Data Words [18, 37]. 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 $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \rightarrow \mathbf{R} \cup \{\text{in}, \perp\}) \times Q$ is a *transition relation* such that if $\tau: (q, a, g^-, g^\#, up, s) \in \Delta$, then $g^- \cap g^\# = \emptyset$. We use $q - \boxed{a \mid g^-, g^\#, up} \rightarrow s$ to denote τ (and often drop from up mappings $r \mapsto r$ for $r \in \mathbf{R}$, which we treat as implicit). The semantics of τ 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 and the $\mathbb{D}$ -value at the current position is equal to all registers from g^- and not equal to any register from $g^\#$; 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 $\perp$).

A *configuration* of $\mathcal{A}$ is a pair $c \in Q \times (\mathbf{R} \rightarrow \mathbb{D} \cup \{\perp\})$, i.e., it consists of a state and an assignment of data to registers. An *initial configuration* of $\mathcal{A}$ is a pair $c_{init} = (q_{init}, \{r \mapsto \perp \mid r \in \mathbf{R}\})$ with $q_{init} \in I$. 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 $\tau: q - \boxed{a \mid g^-, g^\#, up} \rightarrow s \in \Delta$, denoted as $c_1 \xrightarrow{\langle a, d \rangle}_\tau c_2$, iff

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

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 $\rho = c_0 \tau_1 c_1 \tau_2 \dots \tau_n c_n$ such that $\forall 1 \leq i \leq n: c_{i-1} \xrightarrow{\langle a_i, d_i \rangle}_{\tau_i} c_i$ and $c_0 = c$. We say that ρ is *accepting* if c is an initial configuration, $c_n = (s, f)$, and $s \in F$. The *language* $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathcal{A} \text{ has an accepting run over } w\}$.

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 $q - \boxed{a \mid g_1^-, g_1^\#, up_1} \rightarrow s_1, q - \boxed{a \mid g_2^-, g_2^\#, up_2} \rightarrow s_2 \in \Delta$ we have that $g_1^- \cap g_2^\# \neq \emptyset$ or $g_2^- \cap g_1^\# \neq \emptyset$. $\mathcal{A}$ is *complete* if for all states $q \in Q$, symbols $a \in \Sigma$, and $g \subseteq \mathbf{R}$, there is a transition $q - \boxed{a \mid g^-, g^\#, up} \rightarrow s$ such that $g^- \subseteq g$ and $g \cap g^\# = \emptyset$.

Universal RAs. A *universal RA* (URA) $\mathcal{A}_U$ is defined exactly as an NRA with the exception of its language. The language of $\mathcal{A}_U$ is the set $\mathcal{L}(\mathcal{A}_U) = \{w \in (\Sigma \times \mathbb{D})^* \mid \text{every run of } \mathcal{A}_U \text{ on } w \text{ is accepting}\}$ (we emphasize that if a run cannot continue from some state over the current input symbol, then it is not accepting). The concept of *universality* in the name is linked to the use of universal quantification over runs in deciding acceptance.

There is indeed duality between NRAs and URAs, as stated by the following fact.

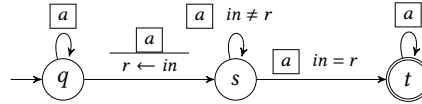
FACT 1. *For every NRA $\mathcal{A}_N$, there is a URA accepting the complement of $\mathcal{L}(\mathcal{A}_N)$. Conversely, for every URA $\mathcal{A}_U$, there is an NRA accepting the complement of $\mathcal{L}(\mathcal{A}_U)$.*

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. $\square$

Example 2.1. Consider the language of words over $\Sigma = \{a\}$ that contain two occurrences of some data value, i.e., the language

$$L_{\exists repeat} = \{w \mid \exists i, j: i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\}.$$

An NRA recognising this language is in the following figure:



Formally, it is an NRA $\mathcal{A} = (\{q, s, t\}, \{r\}, \Delta, \{q\}, \{t\})$ where the transition relation is defined as $\Delta = \{q \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q, q \xrightarrow{a \mid \emptyset, \emptyset, \{r \mapsto in\}} s, s \xrightarrow{a \mid \emptyset, \{r\}, \emptyset} s, s \xrightarrow{a \mid \{r\}, \emptyset, \emptyset} t, t \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} t\}$ (actually, the guard $in \neq r$ on the self-loop over s is redundant). Recall that $\emptyset$ for an update denotes the mapping $\{r \mapsto r\}$. We note that $L_{\exists repeat}$ is not expressible by any DRA or URA.³ Intuitively, $\mathcal{A}$ waits in q until it nondeterministically guesses the input data value that should be repeated, stores it into register r , and moves to state s . In state s , it is waiting to see the data value again, upon which it moves to the accepting state t and reads out the rest of the word.

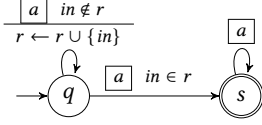
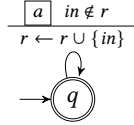
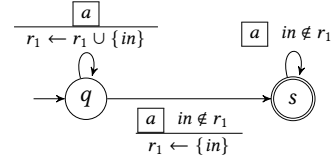
On the other hand, the complement of the language, i.e., the language of words where no two positions have the same data value, formally,

$$L_{\neg \exists repeat} = \{w \mid \forall i, j: i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\},$$

is not expressible by any DRA or NRA [37, Proposition 5 and its proof], but is expressible by a URA. The URA accepting $L_{\neg \exists repeat}$ looks similar to the NRA $\mathcal{A}$ above with the exception of final states, which are $\{q, s\}$ (note that in URAs, in order to accept a word, all runs over the word need to accept, so in order to accept in this example, all runs of the URA need to avoid the state t). $\square$

We use $\text{NRA}^\neq$, $\text{URA}^\neq$, and $\text{DRA}^\neq$ to denote the sub-classes of NRAs, URAs, and DRAs with *no disequality* guards, i.e., automata where for every transition $q \xrightarrow{a \mid g^\neq, g^\neq, up} s$ it holds that $g^\neq = \emptyset$. Furthermore, for a class C of automata with registers and $n \in \mathbb{N}$, we use C_n to denote the sub-class of C containing automata with at most n registers (e.g., DRA_2). We abuse notation and use C to also denote the class of languages defined by C .

³This can be shown by contradiction, assuming that there is a DRA with k registers accepting $L_{\exists repeat}$. One can then take a run over some rejected word of the length $2k + 2$, cut it in half, and modify the second half of the word to contain some symbol that is in the first half of the word but is not stored any register. Because of determinism, the DRA should also reject the modified word, but then the accepted language is not $L_{\exists repeat}$. Contradiction.

Fig. 1. DRSA₁ for $L_{\exists repeat}$ Fig. 2. DRSA₁ for $L_{\neg \exists repeat}$ Fig. 3. RSA₁ for $L_{\neg \forall repeat}$

3 Register Set Automata

On this section, we introduce the model of register set automata, which differ from RAs mainly in the ability to store into registers *sets of values* instead of just single data elements. 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 $Q, \mathbf{R}, I, F$ are the same as for RAs and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \rightarrow 2^{\mathbf{R} \cup \{in\}}) \times Q$ such that if $q - (a | g^{\in}, g^{\notin}, up) \rightarrow s \in \Delta$, then $g^{\in} \cap g^{\notin} = \emptyset$ (as with NRAs, we often do not write mappings $r \mapsto \{r\}$ for $r \in \mathbf{R}$ when defining up). The semantics of a transition $q - (a | g^{\in}, g^{\notin}, up) \rightarrow s$ is that $\mathcal{A}_S$ can move from state q to state s if the Σ -symbol at the current position of the input word is a and the $\mathbb{D}$ -value at the current position is in all registers from $g^{\in}$ and in no register 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).

A *configuration* of $\mathcal{A}_S$ is a pair $c \in Q \times (\mathbf{R} \rightarrow 2^{\mathbb{D}})$, i.e., it consists of a state and an assignment of sets of data values to registers. An *initial configuration* of $\mathcal{A}_S$ is a pair $c_{init} = (q_{init}, \{r \mapsto \emptyset \mid r \in \mathbf{R}\})$ with $q_{init} \in I$. Let $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ be 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 $t: q - (a | g^{\in}, g^{\notin}, up) \rightarrow s \in \Delta$, denoted as $c_1 \vdash_t^{(a,d)} c_2$, 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 $f_2(r) = \bigcup \{f_1(r') \mid r' \in \mathbf{R}, r' \in up(r)\} \cup \begin{cases} \{d\} & \text{if } in \in up(r) \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$

The definition of a run and language of $\mathcal{A}_S$ is then the same as for NRAs.

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 $q - (a | g_1^{\in}, g_1^{\notin}, up_1) \rightarrow s_1, q - (a | g_2^{\in}, g_2^{\notin}, up_2) \rightarrow s_2 \in \Delta$ we have that $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_2^{\in} \cap g_1^{\notin} \neq \emptyset$.

Example 3.1. A DRSA accepting the language $L_{\exists repeat}$ from Example 2.1 is in Fig. 1. Formally, it is a DRSA₁ $\mathcal{A} = (\{q, s\}, \{r\}, \Delta, \{q\}, \{s\})$ where $\Delta = \{q - (a | \emptyset, \{r\}, \{r \mapsto \{r, in\}\}) \rightarrow q, q - (a | \{r\}, \emptyset, \emptyset) \rightarrow s, s - (a | \emptyset, \emptyset, \emptyset) \rightarrow s\}$. 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. $\square$

Example 3.2. A DRSA₁ accepting the language $L_{\neg \exists repeat}$ from Example 2.1 is in Fig. 2. Intuitively, the automaton stays in state q and accumulates input data values in register r , making sure the input data value has not been seen previously. $\square$

Example 3.3. Consider the following language:

$$L_{\neg \forall repeat} = \{w \mid \exists i \forall j: i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\}$$

Intuitively, $L_{\neg \forall repeat}$ is the language of all words containing a data value with exactly one occurrence. This language is accepted, e.g., by the RSA₁ in Fig. 3. The RSA stays in state q , collecting the seen values into its register, and at some point, when it sees a value not seen previously, it

nondeterministically moves to s , remembering the value in its register. Then, at state s , the RSA just checks that it does not see the remembered value any more. We note that $L_{\neg \forall \text{repeat}}$ cannot be accepted by a DRSA (cf. the proof of Theorem 4.6). $\square$

4 Properties of Register Set Automata

In this section, we establish decidability and complexity of basic decision problems for RSAs and their closure properties. First, we claim that RSAs generalise NRAs.

FACT 2. *For every $n \in \mathbb{N}$ and NRA_n , there exists an RSA_n accepting the same language.*

The next theorem shows the core property of RSAs: that their emptiness problem is decidable, however, for a much higher price than for NRAs, for which it is PSPACE-complete⁴ [18]. For classifying the complexity of the problem, we use the hierarchy of fast-growing complexity classes of Schmitz [57], in particular the class F_ω , which, intuitively, corresponds to Ackermannian problems closed under primitive-recursive reductions.

THEOREM 4.1. *The emptiness problem for RSA is decidable, in particular, F_ω -complete.*

SKETCH OF 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 F_ω -complete problem [58–60]. In the following, we briefly describe both directions of the reduction (see [30] for details and examples).

(RSA emptiness $\leq$ TPN coverability) Intuitively, the conversion of an RSA $\mathcal{A} = (Q, R, \Delta, I, F)$ 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: (i) one place for each state of $\mathcal{A}$, (ii) two special places *init* and *fin*, and (iii) one place for every subset $\text{rgn} \subseteq R$; these places are used to represent all possible intersections of values held in registers. E.g., 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. 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 $\text{rgn} \in 2^R$. 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 nondeterministically chooses one state from I) and the place *fin* is used as the coverability test target; all places corresponding to final states of $\mathcal{A}$ can simply transition into it.

(TPN coverability $\leq$ RSA emptiness) 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 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

⁴Note that for an alternative definition of NRAs considered in [37, 55], where no two registers can contain the same data value, the problem is NP-complete [55].

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. $\square$

Remark 1. Since RSAs generalise NRAs, their universality, equivalence, and language inclusion problems are all undecidable.

4.1 Closure Properties

The closure properties of RSAs under Boolean operations are the same as for NRAs (cf. [37, Proposition 5, Theorem 3]).

THEOREM 4.2. *The following closure properties hold for the class RSA:*

- (1) RSA is closed under union and intersection.
- (2) RSA is not closed under complement.

SKETCH OF PROOF. The proofs for closure under union and intersection are standard. For showing the non-closure under complement, consider the language $L_{\neg\forall repeat}$ from Example 3.3, which can be accepted by RSA. We use a similar technique as in the proof of Proposition 3.2 in [23] and show that if there were an RSA accepting its complement, namely, the language

$$L_{\forall repeat} = \{w \mid \forall i \exists j: i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\},$$

RSAs could be used to decide the emptiness problem of a Minsky machine, which is a known undecidable problem. See the full proof in [30] for more details. $\square$

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

THEOREM 4.3. *For each $n \in \mathbb{N}$, the class RSA_n is closed under union and not closed under intersection and complement.*

SKETCH OF PROOF. The proof of closure of RSA_n under union is standard. For proving non-closure of RSA_1 under intersection, we consider the two following RSA_1 languages

$$\mathcal{L}_1^A = \{w \mid \mathbb{D}[w_1] = \mathbb{D}[w_{|w|}]\} \quad \text{and} \quad \mathcal{L}_1^B = \{w \mid \mathbb{D}[w_2] = \mathbb{D}[w_{|w|-1}]\} \quad (1)$$

and show that their intersection cannot be accepted by RSA_1 . This argument can be extended to RSA_n for $n > 1$. Non-closure of RSA_n under complement then follows from De Morgan's laws. $\square$

THEOREM 4.4. *DRSA is closed under union, intersection, and complement.*

PROOF. The proofs are standard (product construction and swapping (non)-final states). $\square$

THEOREM 4.5. *For each $n \in \mathbb{N}$, the class $DRSA_n$ is closed under complement and not closed under union and intersection.*

PROOF. Proof of closure under complement is standard. Non-closure under intersection is done similarly as in the proof of Theorem 4.3. Non-closure for union follows from De Morgan's laws. $\square$

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

THEOREM 4.6. $DRSA \subseteq RSA$

Algorithm 1: Determinization of an NRA into a DRSA

Input : Copyless NRA $\mathcal{A} = (Q, R, \Delta, I, F)$
Output : DRSA $\mathcal{A}' = (Q', R, \Delta', I', F')$ with $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ or $\perp$

- 1 $Q' \leftarrow \text{worklist} \leftarrow I' \leftarrow \{(I, c_0 = \{r \mapsto 0 \mid r \in R\})\};$
- 2 $\Delta' \leftarrow \emptyset;$
- 3 **while** worklist $\neq \emptyset$ **do**
- 4 $(S, c) \leftarrow \text{worklist.pop}();$
- 5 **foreach** $a \in \Sigma, g \subseteq \{r \in R[S] \mid c(r) \neq 0\}$ **do**
- 6 $T \leftarrow \{q \mid \boxed{a \mid g^-, g^\#, \cdot} \rightarrow q' \in \Delta \mid q \in S, g^- \subseteq g, g^\# \cap g = \emptyset\};$
- 7 $S' \leftarrow \{q' \mid \boxed{\cdot \mid \cdot, \cdot, \cdot} \rightarrow q' \in T\};$
- 8 **if** $\exists q \mid \boxed{\cdot \mid \cdot, g^\#, \cdot} \rightarrow q' \in T, \exists r \in g^\# : c(r) > 1$ **then return** $\perp$;
- 9 $T^\bullet = \{q \mid \boxed{a \mid g^-, g^\#, \text{up}[g^- \mapsto \text{in}]} \rightarrow q' \mid q \mid \boxed{a \mid g^-, g^\#, \text{up}} \rightarrow q' \in T\};$
- 10 **foreach** $r_i \in R$ **do** // update the register size classes
- 11 $\text{tmp} \leftarrow \emptyset;$
- 12 **foreach** $\boxed{\cdot \mid g^-, \cdot, \text{up}} \rightarrow \cdot \in T^\bullet$ **do**
- 13 **if** $\text{up}(r_i) = y \neq \perp \wedge c(y) \neq 0$ **then** $\text{tmp} \leftarrow \text{tmp} \cup \{y\}$;
- 14 $\text{op}_{r_i} \leftarrow \text{tmp};$
- 15 $c'(r_i) \leftarrow \sum_{x \in \text{op}_{r_i}} c(x) ;$ $> 1 \rightsquigarrow \omega$
- 16 **foreach** $q' \in S'$ **do** // check for Cartesian overapproximation
- 17 $P \leftarrow \text{op}_{r_1} \times \dots \times \text{op}_{r_n}$ for $\{r_1, \dots, r_n\} = R[q']$;
- 18 **foreach** $(x_1, \dots, x_n) \in P$ **do**
- 19 **if** $\nexists (\boxed{\cdot \mid \cdot, \cdot, \text{up}} \rightarrow q') \in T^\bullet$ s.t. $\bigwedge_{1 \leq i \leq n} \text{up}(r_i) = x_i$ **then return** $\perp$;
- 20 $\text{up}' \leftarrow \{r_i \mapsto \text{op}_{r_i} \mid r_i \in R\};$
- 21 **if** $(S', c') \notin Q'$ **then**
- 22 $\text{worklist.push}((S', c'));$
- 23 $Q' \leftarrow Q' \cup \{(S', c')\};$
- 24 $\Delta' \leftarrow \Delta' \cup \{(S, c) \mid \boxed{a \mid g, R \setminus g, \text{up}'} \rightarrow (S', c')\};$
- 25 **return** $\mathcal{A}' = (Q', R, \Delta', I', \{(S, c) \in Q' \mid S \cap F \neq \emptyset\})$;

PROOF. Let us consider the language $L_{\neg \forall \text{repeat}}$ from the proof of Theorem 4.2, which is expressible using RSAs, and its complement $L_{\forall \text{repeat}}$, which is not expressible using RSAs. Since DRSA are closed under complement (Theorem 4.4), if they could accept $L_{\neg \forall \text{repeat}}$, they could also accept $L_{\forall \text{repeat}}$, which is a contradiction. Therefore, $L_{\neg \forall \text{repeat}} \notin \text{DRSA}$. $\square$

4.2 Expressivity

The RSA model captures an interesting class of data word languages, strictly generalizing NRAs and being incomparable to ARAs [18] or pebble automata [48]. Due to the page limit, see [30] for detailed positioning of RSAs in the landscape of register automata models.

5 Determinizing Register Automata

RSAs have the following interesting property: a large class of NRAs can be determinized into DRSA (we emphasize that the determinization considered here changes the model from one storing single values in registers (NRAs) to one storing sets of values in registers (DRSAs)). In this section, we give a determinization semi-algorithm and specify properties of a class of NRAs for which it is complete.

Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NRA. We use $R[q]$ for $q \in Q$ to denote the set of registers r *active* at q , for which there exists a transition $s \xrightarrow{(\cdot \mid g^{\neq}, g^{\neq}, up)} t \in \Delta$ with (i) $up(r) \neq \perp$ and $t = q$ or (ii) $r \in g^{\neq} \cup g^{\neq}$ and $s = q$. Besides being the basis for the register locality optimization in Section 5.1.2, the set of active registers $R[q]$ is also used in the basic algorithm as an overapproximation of the set of registers with a value different from $\perp$ (it excludes those register that were just assigned $\perp$). Given a set of states S , we define $R[S] = \bigcup_{q \in S} R[q]$. Furthermore, we call $\mathcal{A}$ *copyless* if there is no reachable configuration (q, f) such that $f(r_1) = f(r_2) \neq \perp$ for a pair of distinct registers $r_1, r_2 \in R$, i.e., there is at most one copy of each data value in $\mathcal{A}$. Again, any NRA can be converted into the copyless form, however, the number of states can increase to $B_{|R|} \cdot |Q|$ where B_n is the n -th Bell number. Intuitively, the transformation is done by creating one copy of each state for every possible partition of R (the partitions contain registers with the same value), and modifying the transition function correspondingly.

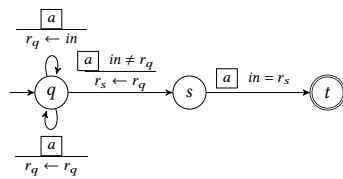
The determinization (semi-)algorithm for a copyless NRA $\mathcal{A}$ is shown in Algorithm 1. On the high level, it is similar to the classical Rabin-Scott subset construction for determinizing finite automata [52] with additional treatment of registers superimposed onto it.

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) a mapping $c: R \rightarrow \{0, 1, \omega\}$ assigning to each register its *size class*, that records whether the size of the set in the register is 0, 1, or larger than 1 (denoted by the ω). The size classes are needed to ensure that our simulation of a disequality test $in \neq r$ by the non-membership test $in \notin r$ is precise, as explained under the item (1) below. The macrostate is then a pair (S, c) . 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).

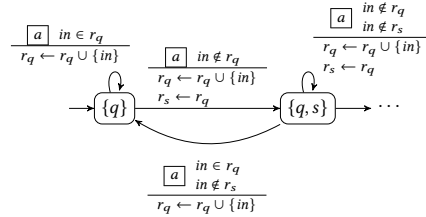
The main loop of the algorithm then constructs successors of reachable macrostates for each $a \in \Sigma$ and each $g \subseteq R$ on Line 5; each pair a, g corresponds to the so-called *minterm* (minterms denote combinations of guards whose semantics do not overlap [14]). The set of active registers $R[S]$ is used here to prune those minterms that clearly cannot be satisfied, since they are testing a register whose value must be $\perp$. 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 10–20).

The algorithm uses the following three techniques to handle three sources of imprecision:

- (1) *Register size classes.* (Lines 10 to 15). 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 $g^{\neq}$, 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 following example:



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

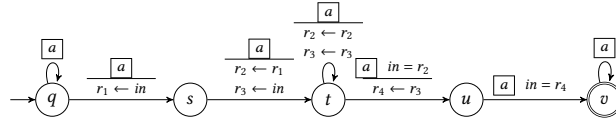


(b) A part of the DRSA obtained for $\mathcal{A}$

where (b) contains a part of the DRSA obtained if Algorithm 1 did not use the c -component of macrostates. The reason for this is that after reading the third symbol (i.e., $\langle a, 2 \rangle$), the RSA

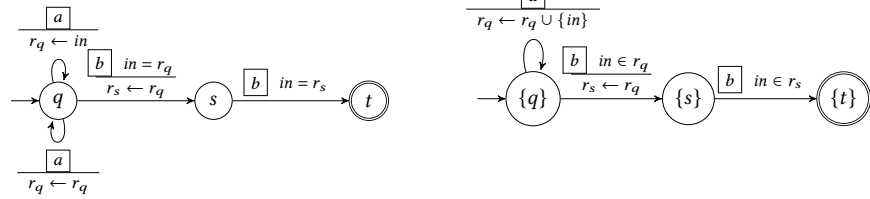
goes to the macrostate $\{q\}$ —it thinks it cannot be in s any more. This is the reason why we augment macrostates with the c -component, the register size classes. They allow to detect that a disequality test is performed on a register containing more than one element, in which case we terminate the algorithm (Line 8). The sizes classes are updated on Line 15 where the sum is *saturated* to ω for values > 1 (denoted by $\sum^{>1 \leadsto \omega}$).

- (2) *Checking for Cartesian overapproximation* (Lines 16 to 19). By collecting all possible values that can occur in registers, the algorithm is performing the so-called *Cartesian overapproximation* (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 $\{uvwvz \mid u, w, z \in (\Sigma \times \mathbb{D})^*, |v| = 2\}$:



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 also 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 16–19. Note that the overapproximation checking is overly conservative in a way that for some automata, if we did not do the check and continued the construction, outputting the DRSA, the resulting DRSA would still be correct (see [30]).⁵

- (3) *Choice collapse at tests* (Line 9). When a set-register has collected several nondeterministic choices of values for a standard NRA register, then testing membership of a concrete value, which stands for testing equality of the NRA register, should collapse the choices to that single value. Without the collapse, a subsequent second membership test with a different value might pass as if the NRA register could hold two different values at once within a single non-deterministic case. Consider the following example of an NRA $\mathcal{A}$ and an RSA obtained from $\mathcal{A}$ by Algorithm 1 without the substitution $up[g^= \mapsto in]$ on Line 9 (to save space, we collapse all macrostates with the same set of states into one):

(a) An NRA $\mathcal{A}$ (b) A DRSA overapproximating $\mathcal{A}$'s language

One can see that while $\mathcal{A}$ cannot accept the word $\langle a, 1 \rangle \langle a, 2 \rangle \langle b, 1 \rangle \langle b, 2 \rangle$, the DRSA can. This happens because the DRSA 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

⁵In the implementation, we actually postpone the overapproximation test only after the whole DRSA is constructed. At this point, we can check more precisely whether there is some real overapproximation.

from $\{q\}$ to $\{s\}$) succeeded. We avoid this situation by the substitution on Line 9, which performs the collapse of the set of nondeterministic choices given by particular values of g^- 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 *copyless*, i.e., it never happens that a data value is in more than one register.

The correctness of Algorithm 1 is summarized by the following theorem, proved in [30].

THEOREM 5.1. *When Algorithm 1 returns a DRSA $\mathcal{A}'$, then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

The following proposition establishes the numbers of states and transitions of the output DRSA of Algorithm 1, which follow directly from the structure of macrostates and transitions of $\mathcal{A}'$.

PROPOSITION 5.2. *Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an RA and let Algorithm 1 return a DRSA $\mathcal{A}' = (Q', R, \Delta', I', F')$. Then $|Q'| \leq 2^{|Q| + (\log_2 3) \cdot |R|}$ and $|\Delta'| \leq |\Sigma| \cdot 2^{|Q| + (\log_2 6) \cdot |R|}$.*

We note that the lower bound on $|Q'|$ is $2^{|Q|}$, which comes from the lower bounds of determinization of finite automata [20, Section 1.4.1]. In the RAs obtained from regexes, typically $|Q| \gg |R|$.

We can also modify Algorithm 1 to omit the c -component of macrostates (S, c) , which will give us the bounds $|Q'| \leq 2^{|Q|}$ (i.e., the same as for finite automata determinization) and $|\Delta'| \leq |\Sigma| \cdot 2^{|Q| + |R|}$. This simplification of the algorithm may in practice restrict the class of input RAs on which it successfully terminates. This is, however, not an issue for RAs *without* disequality guards $g^\#$, which are output by our algorithm for converting regexes to RAs (cf. Section 6.1); for those, the modified algorithm successfully terminates in the same cases (the size of the result might, however, be different in both directions—on the one hand, having the c -component increases the maximum possible number of states, but on the other hand, keeping track of which registers are empty can avoid generation of transitions and states that can never be used in a run).

5.1 Improvements

In this section, we propose improvements of the determinization algorithm. In particular, we introduce slight modifications of Algorithm 1 as well as additional preprocessing of input NRAs aiming at enlarging the class of NRAs that can be determinized to DRSA

5.1.1 Refining the Register Size Map. One approach to enlarge the class of NRAs that can be determinized by Algorithm 1 is to keep the value of c of each macrostate that overapproximate the register size as precise as possible. Indeed, the value of c affects the condition on Line 8. A way how c might become unnecessarily high is when $tmp \cap g \neq \emptyset$ and $in \in tmp$. In that case, $c(in) = 1$ does not need to be added to c' since according to the transition guard we know that in is in g . We hence modify the Line 14 to

$$op_{r_i} \leftarrow \begin{cases} tmp \setminus \{in\} & \text{if } tmp \cap g \neq \emptyset \text{ and} \\ tmp & \text{otherwise} \end{cases}$$

5.1.2 Register Locality. Another feature limiting the determinizability of the given NRA are nondeterministic transitions dealing with the same registers violating the Cartesian overapproximation checked on Line 19. In order to increase the locality of registers, we propose register-local NRAs. Locality of registers limiting updates of the same registers in different states and hence reducing the possibility of the condition on Line 19 holds true. Formally, we call $\mathcal{A}$ *register-local* if for all

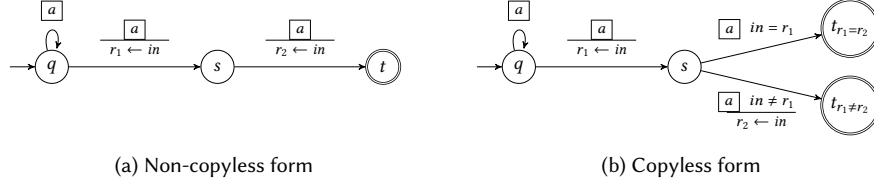


Fig. 4. Copyless conversion for an NRA that stores data values in two registers.

$r \in \mathbf{R}$ 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 $|Q| \cdot |\mathbf{R}|$. Formally, given a NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$, we define its *register-localization* as $\mathcal{A}_\bullet = (Q, \mathbf{R}_\bullet, \Delta_\bullet, I, F)$ where $\mathbf{R}_\bullet = \{r_q \mid q \in Q, r \in \mathbf{R}\}$ and $\Delta_\bullet$ is defined as follows:

$$\Delta_\bullet = \left\{ p \xrightarrow{a \mid g_\bullet^-, g_\bullet^\#, up_\bullet} q \mid p \xrightarrow{a \mid g^-, g^\#, up} q \in \Delta, \quad g_\bullet^- = v_p(g^-), \quad g_\bullet^\# = v_p(g^\#), \right. \\ \left. up_\bullet = \{(v_q(r), v_p(t)) \mid (r, t) \in up\} \cup \{(v_s(r), \perp) \mid s \in Q \setminus \{q\}, r \in \mathbf{R}\} \right\} \quad (2)$$

where the localization function v_q is defined as $v_q(x) = x_q$ if $x \in \mathbf{R}$, otherwise $v_q(x) = x$ for $x \in \{in, \perp\}$. We extend the definition of the localization function to set of values in the usual way. As a prior step to Algorithm 1, we first convert input NRA to its register-local form.

5.1.3 Relaxing the Copyless Property. Conversion of an input NRA to an equivalent copyless NRA may introduce non-equality guards on registers having nondeterministically chosen values (i.e., in two runs over the same string, the registers can have different values at the same position in the string). Such non-equality guards can cause Algorithm 1 to return $\perp$ on Line 8 as it is shown in Example 5.3. In order to reduce the introduction of non-equality guards, we relax the copyless condition into *relation-free* condition in a way that there is no transition where multiple registers are assigned the same value regardless of the input. In other words, registers can hold the same value as long a run does not induce equality relations on the register values. Syntactically, relation-freeness can be expressed as each register must appear on a right-hand side of an update at most once, and at most one register can be updated by *in* or a register in the equality guard. Note that every NRA can be converted to a relation-free form, which is then used as an input of Algorithm 1.

Example 5.3. Consider the NRA over $\Sigma = \{a\}$ shown in Fig. 4a. It non-deterministically selects a data value to store in r_1 and then stores the following data value in r_2 . Equivalent copyless NRA is shown in Fig. 4b. The NRA non-deterministically selects the data value for r_1 as well, but before storing the next data value in r_2 , it must check that the data value is not already stored in r_1 . If the data value is already stored in r_1 , then it marks that $r_1 = r_2$ in its state control and does not actually store anything in r_2 . With this construction, however, we necessarily introduce a non-equality guard on r_1 , which eventually causes Algorithm 1 to return $\perp$ on Line 8.

5.2 Determinizability

Naturally, we wish to syntactically characterise the class of NRAs for which Algorithm 1 is complete. We observe that when we start with an NRA_1^- and apply the register localization the algorithm always returns a DRSA (the theorem is proved in [30]).

THEOREM 5.4. *For every NRA_1^- , there exists a DRSA accepting the same language.*

Let $\mathcal{B}(\text{NRA}_1^-)$ be the class of languages that can be expressed using a Boolean combination of NRA_1^- languages, i.e., it is the closure of NRA_1^- languages under union, and intersection, and complement (it could also be denoted as $\mathcal{B}(\text{URA}_1^-)$).

Example 5.5. For instance, the language $L_{\exists, \neg \exists \text{repeat}}$ composed as the concatenation of $L_{\exists \text{repeat}}$ and $L_{\neg \exists \text{repeat}}$ with a delimiter, formally

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

is in $\mathcal{B}(\text{NRA}_1^-)$, since it is the intersection of languages

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

but is expressible neither by an NRA nor by a URA (URAs cannot express the part *before* the delimiter and NRAs cannot express the part *after* the delimiter). $\square$

The determinizability of $\mathcal{B}(\text{NRA}_1^-)$ then follows directly from Theorems 4.4 and 5.4.

COROLLARY 5.6. *For any language in $\mathcal{B}(\text{NRA}_1^-)$, there exists a DRSA accepting it.*

A direct consequence of Corollary 5.6 and decidability of the emptiness check is also decidability of the inclusion problem as it is shown in the following corollary.

COROLLARY 5.7. *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 5.6 and Theorems 4.1 and 4.2. $\square$

6 Matching of Regexes with Backreferences

In this chapter, we define regexes with backreferences and show how they can be converted into NRAs using a construction based on Antimirov derivatives. We then establish the complexity of regex matching with DRSA's constructed from these regexes.

6.1 Regexes with Backreferences

Syntax. A *regex with backreferences* (REBR) $\mathcal{R}$ over alphabet Σ is defined inductively according to the following grammar:

$$\mathcal{R} ::= \epsilon \mid S \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R}^* \mid (S)_m \mid \backslash m$$

where $S \subseteq \Sigma$, and $m \in \mathbb{N}$. In addition to the standard regex syntax, the definition introduces (i) a *capture group* $(S)_m$ (explicitly indexed by m), capturing a single symbol from the set S , and (ii) a *backreference* $\backslash m$ to the m -th capture group. We use RE for the set of all REBRs. Note that compared to the languages that can be specified using, e.g., *Perl Compatible Regular Expressions* (PCREs) [32], our definition does not enable expressing languages where capture groups have unbounded length, e.g., the language defined by the PCRE $/^{\wedge}(.*)\backslash 1\$/$, which matches strings of the form ww . Capture groups with bounded length (but longer than 1) can be expressed using REBRs by splitting the longer capture groups into single-letter ones, e.g., PCRE $/(\dots)\backslash 1/$ can be expressed as $/(\cdot)(\cdot)(\cdot)\backslash 1\backslash 2\backslash 3/$.

Semantics. We will define the semantics of REBR $\mathcal{R}$ in two phases: 1) in terms of *annotated words*, with marked parts matched by capture-groups and “pointers” to capture groups in place of backreferences; and 2) normal words obtained by interpreting the annotations.

In phase 1, subwords matching capture groups are annotated by the group’s index m and the keyword `in`, and backreference annotations consist of the index of a capture group and the keyword `ref`. The language of the annotated words $\mathcal{L}_{\text{ref}}$ is hence a language over the alphabet $\Sigma_{\text{ref}} = \Sigma \cup \{\langle s, m, \text{in} \rangle, \langle m, \text{ref} \rangle \mid s \in \Sigma, 0 \leq m \leq k_{\mathcal{R}}\}$, with $k_{\mathcal{R}}$ being the maximum index of a capture group or a backreference in $\mathcal{R}$, defined as follows:

$$\begin{aligned} \mathcal{L}_{\text{ref}}(\epsilon) &= \{\epsilon\} & \mathcal{L}_{\text{ref}}(\mathcal{R}_1^*) &= (\mathcal{L}_{\text{ref}}(\mathcal{R}_1))^* \\ \mathcal{L}_{\text{ref}}(S) &= S & \mathcal{L}_{\text{ref}}((S)_m) &= \{\langle s, m, \text{in} \rangle \mid s \in S\} \\ \mathcal{L}_{\text{ref}}(\mathcal{R}_1 + \mathcal{R}_2) &= \mathcal{L}_{\text{ref}}(\mathcal{R}_1) \cup \mathcal{L}_{\text{ref}}(\mathcal{R}_2) & \mathcal{L}_{\text{ref}}(\backslash m) &= \{\langle m, \text{ref} \rangle\} \\ \mathcal{L}_{\text{ref}}(\mathcal{R}_1 \cdot \mathcal{R}_2) &= \mathcal{L}_{\text{ref}}(\mathcal{R}_1) \cdot \mathcal{L}_{\text{ref}}(\mathcal{R}_2) \end{aligned}$$

Intuitively, a backreference should match the character that the corresponding capture group matched last in the prefix left of the backreference. Formally, for the m -th capture group and a prefix w , it is the character $\text{match}(w, m) = c$ if it is possible to write w as $w_p \cdot \langle c, m, \text{in} \rangle \cdot w_s$ where $w_p \in \Sigma_{\text{ref}}^*$ and $w_s \in \Sigma_{\text{ref}}^*$ does not contain *any occurrence* of a symbol $\langle a, m, \text{in} \rangle$ for arbitrary $a \in \Sigma$ (hence this split of the prefix w indeed marks the last match of the capture group in it). Otherwise, if w has no occurrence of an annotated capture group match $\langle a, m, \text{in} \rangle$, then $\text{match}(w, m) = \perp$.

In order to obtain a language over the alphabet Σ from $\mathcal{L}_{\text{ref}}$, we define *annotation interpretation*, denoted as interp , which replaces the backreference annotations with the captured characters and removes the capture group annotations. Formally, given a word $w \in \Sigma_{\text{ref}}^*$ and an index $1 \leq i \leq |w|$, interp is defined as

$$\text{interp}(w, i) = \begin{cases} \text{match}(w[1 : i-1], m) & \text{if } w[i] = \langle m, \text{ref} \rangle, \\ c & \text{if } w[i] = \langle c, m, \text{in} \rangle, \\ w[i] & \text{otherwise,} \end{cases}$$

where $w[1 : i] = w_1 \dots w_i$ for $w = w_1 \dots w_{|w|}$. We lift the definition to a word as $\text{interp}(w) = \{w' \mid w' = \text{interp}(w, 1) \dots \text{interp}(w, |w|), w' \text{ does not contain } \perp\}$. Notice that removing the words containing $\perp$ removes exactly cases where a backreference occurs before the corresponding capture group is matched. The *language* of a REBR $\mathcal{R}$ is then defined as $\mathcal{L}(\mathcal{R}) = \bigcup_{w \in \mathcal{L}_{\text{ref}}(\mathcal{R})} \text{interp}(w)$. Since our automaton model works with data words and the semantics of REBRs are defined as languages over Σ , we define the extension of words/languages over Σ to data words. For that we consider some *fixed* linear ordering on Σ , meaning that there is an injective mapping $\text{ord} : \Sigma \rightarrow \mathbb{N}$. Then, for a word $w = a_1 \dots a_n \in \Sigma^*$, we define $w^\bullet = \langle a_1, \text{ord}(a_1) \rangle \dots \langle a_n, \text{ord}(a_n) \rangle$. We lift the definition to languages $\mathcal{L}^\bullet$ in the usual way.

Antimirov Derivatives. We can translate REBRs to RAs using a construction based on *Antimirov derivatives* [3]. A *partial derivative* is a function $\partial : \Sigma \times RE \rightarrow 2^{RE \times \mathbb{N}_\perp \times \mathbb{N}_\perp}$ (we use $\mathbb{N}_\perp$ to denote the set $\mathbb{N} \cup \{\perp\}$) such that $\partial_a(\mathcal{R})$ gives us all possible REBRs (together with potential information about writing or testing a register) obtained from $\mathcal{R}$ by trying to match a string with leading symbol $a \in \Sigma$. The definition of the derivatives is in Fig. 5. In the definition, we also use the predicate *nullable*($\mathcal{R}$), which denotes that $\mathcal{R}$ is *nullable*, i.e., it accepts the empty string. We note that our class of REBRs allows this relatively simple extension of Antimirov derivatives; if one considered unbounded capture groups, the derivatives would get more complex.

$$\begin{aligned}
\partial_a(\epsilon) &= \emptyset & \text{nullable}(\epsilon) &\Leftrightarrow \text{true} \\
\partial_a(S) &= \begin{cases} \{\langle \epsilon, \perp, \perp \rangle\} & \text{if } a \in S \\ \emptyset & \text{otherwise} \end{cases} & \text{nullable}(S) &\Leftrightarrow \text{false} \\
\partial_a(r_1 + r_2) &= \partial_a(r_1) \cup \partial_a(r_2) & \text{nullable}(r_1 + r_2) &\Leftrightarrow \text{nullable}(r_1) \vee \text{nullable}(r_2) \\
\partial_a(r_1 \cdot r_2) &= \begin{cases} \alpha_a(r_1 \cdot r_2) \cup \partial_a(r_2) & \text{if } \text{nullable}(r_1) \\ \alpha_a(r_1 \cdot r_2) & \text{otherwise} \end{cases} & \text{nullable}(r_1 \cdot r_2) &\Leftrightarrow \text{nullable}(r_1) \wedge \text{nullable}(r_2) \\
&\text{where } \alpha_a(r_1 \cdot r_2) = \{\langle r \cdot r_2, \text{tst}, \text{up} \rangle \mid \langle r, \text{tst}, \text{up} \rangle \in \partial_a(r_1)\} \\
\partial_a(r^*) &= \{\langle r' \cdot r^*, \text{tst}, \text{up} \rangle \mid \langle r', \text{tst}, \text{up} \rangle \in \partial_a(r)\} & \text{nullable}(r^*) &\Leftrightarrow \text{true} \\
\partial_a((S)_m) &= \{\langle r, \perp, r_m \rangle \mid \langle r, \perp, \perp \rangle \in \partial_a(S)\} & \text{nullable}((S)_m) &\Leftrightarrow \text{false} \\
\partial_a(\backslash m) &= \{\langle \epsilon, r_m, \perp \rangle\} & \text{nullable}(\backslash m) &\Leftrightarrow \text{false}
\end{aligned}$$

Fig. 5. Antimirov derivatives for REBRs. The predicate $\text{nullable}(\mathcal{R})$ outputs true iff $\epsilon \in \mathcal{L}^*(\mathcal{R})$.

NRA Construction. We use the partial derivatives defined above to transform a regex $\mathcal{R}$ into an NRA in the function `rebr2ra`. Let $k_{\mathcal{R}}$ be the maximum index of a capture group or a backreference occurring in $\mathcal{R}$. Then we define the NRA $\text{rebr2ra}(\mathcal{R}) = (Q, \{r_0, \dots, r_{k_{\mathcal{R}}}\}, \Delta, \{\mathcal{R}\}, F)$ where

- the set of states Q contains all regexes that are reachable from $\mathcal{R}$ by the partial derivative construction, i.e., $Q = \mu Z: \bigcup \{q' \mid \langle q', \cdot, \cdot \rangle \in \partial_a(q), q \in Z, a \in \Sigma\} \cup \{\mathcal{R}\} \cup Z$,
- Δ is the smallest set such that if $\langle q', t, u \rangle \in \partial_a(q)$ for $a \in \Sigma$, then $q \xrightarrow{a \mid g^-, \emptyset, \text{up}} q' \in \Delta$ where
 - $g^- = \{r_i\}$ if $u = i \neq \perp$, else $g^- = \emptyset$ and
 - $\text{up} = \{r_i \mapsto r_i \mid 0 \leq i \leq k_{\mathcal{R}}, i \neq j\} \cup \{r_j \mapsto \text{in}\}$ if $u = j \neq \perp$, else $\text{up} = \{r_i \mapsto r_i \mid 0 \leq i \leq k_{\mathcal{R}}\}$, and
- $F = \{q \in Q \mid \text{nullable}(q)\}$.

Note that the number of used registers matches the number of capture groups/backreferences in $\mathcal{R}$. The argument that the number of states of $\text{rebr2ra}(\mathcal{R})$ is finite is similar as the one in [3]; more precisely, the number of states of the output NRA is in $O(|\mathcal{R}|)$. Then, using the properties of Antimirov derivatives and the construction described above, we have the following theorem.

THEOREM 6.1. *For a regex $\mathcal{R}$ we have $\mathcal{L}(\text{rebr2ra}(\mathcal{R})) = \llbracket \mathcal{R} \rrbracket^*$.*

6.2 Regex Matching

In order to match a word $w \in \Sigma^*$ w.r.t. a given REBR $\mathcal{R}$, we first construct an RA $\mathcal{A}$ corresponding to $\mathcal{R}$ (using `rebr2ra`). Then, after register localization, we use Algorithm 1 to determinize $\mathcal{A}$.⁶ If the algorithm does not return $\perp$, we have a DRSA $\mathcal{A}'$ representing $\mathcal{R}$. When testing regex membership, we check that $w^\bullet \in \mathcal{L}(\mathcal{A}')$ using an algorithm tracking configurations obtained during reading of $w^\bullet$. The following lemma establishing the complexity of this algorithm.

LEMMA 6.2. *Let w be a data word and $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be a DRSA over a finite data domain $\mathbb{D}$. Checking that $w \in \mathcal{L}(\mathcal{A})$ can be done in time $O(|\mathbb{D}| \cdot |w| \cdot |\mathbf{R}|^2)$.*

PROOF SKETCH. Consider a data word w and a DRSA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$. Further let $\mathcal{D} = \{v_i \mid v_1 \dots v_n = \mathbb{D}[w]\}$ (note that $|\mathcal{D}| \leq |\mathbb{D}|$). In this proof and the following text we assume a unit cost

⁶Regarding the determinizability of RAs obtained from REBRs, from Theorem 5.4 and the properties of the RA construction we have that we are able to process arbitrary REBRs containing a single capture group (but possibly multiple backreferences).

of comparison between elements of $\mathbb{D}$. The membership checking algorithm tracks configurations of the form (q, r) where $q \in Q$ and $r: \mathbf{R} \rightarrow 2^{\mathbb{D}}$. We assume that set-register values are stored in a sparse-set data structure with constant insertion and membership and with the union of two sets linear to the size of one of the two sets (elements of one sets are inserted to the other, creation of a new sparse-set is not needed as the number of registers is fixed). When constructing a next-state configuration, we first need to check transition guards to get a successor transition. Here, we assume that the successor transitions are kept in a BDD-like structure. Hence, for each register, it suffices to check whether the input symbol is in the register, and then use this bitvector to find an appropriate transition (recall the automaton is deterministic). This can be done in $O(|\mathbf{R}|)$. Then, we need to compute the transition update, which can be done in $O(|\mathbb{D}| \cdot |\mathbf{R}|^2)$. Therefore the overall complexity is $O(|\mathbb{D}| \cdot |w| \cdot |\mathbf{R}|^2)$. $\square$

Note that if the data domain is infinite, testing $w \in \mathcal{L}(\mathcal{A})$ can be done in time $O(|w|^2 \cdot |\mathbf{R}|^2)$. Directly from Lemma 6.2 we have the following theorem.

THEOREM 6.3. *For a fixed REBR $\mathcal{R}$ over a fixed alphabet, the worst-case time complexity of DRSA-based regex matching is linear to the length of the input word (provided Algorithm 1 finishes on the RA obtained from $\mathcal{R}$).*

6.3 Finer Complexity Analysis

The constant character cost complexity of matching in Theorem 6.3 hides a quadratic dependency on the number of registers $|\mathbf{R}|$ and the linear dependency on the size of the data domain $|\mathbb{D}|$. When the DRSA is obtained from a REBR of the size m and with r backreferences by (i) our derivative-based construction, (ii) register localization, and (iii) our determinization (Algorithm 1), then we have $|\mathbf{R}| \in O(m \cdot r)$. The $|\mathbb{D}|$ particularly can be large in practice and problematic.

We will argue that the factor $|\mathbb{D}|$, coming from the need to unite and copy registers, can be avoided if the automaton never copies registers. Copying is thus never done and the cost of uniting registers is covered by the maximum cost of adding elements to the registers. The quadratic dependence on $|\mathbf{R}| = rm$ also decreases to linear. Formally, we define an RSA as *copy-free* if it has no transition where it assigns the same non-singleton register to two registers. Practical relevance of this class is witnessed by the fact that in our experiments, the copy-free DRSA were constructed in 488 cases out of 1,335 (our main benchmark set of single-letter regexes that were found ReDoS prone).

THEOREM 6.4. *For a copy-free DRSA, the complexity of testing membership is $O(|\mathbf{R}| \cdot |w|)$.*

PROOF SKETCH. Let us measure the age of a data value by when it was read from the input, and let the age of a register be the age of its oldest value. Let us union two registers by inserting values from the younger one to the older register. With sparse-sets, the complexity would be linear to the size of the younger register, as every insertion is constant-time. That is, within one union operation, each element in the younger register contributes $O(1)$. Note that since the automaton is copy-free, a value taken from a position in the input data word can be this way (within the union operation) inserted into older registers at most $|\mathbf{R}|$ times (after $|\mathbf{R}|$ unions, the value must be in the oldest register). Hence, inserting the value from a particular position in the input word within unions may take at most $O(|\mathbf{R}|)$ time, and we have $|w|$ positions, which amounts to $O(|\mathbf{R}| \cdot |w|)$ for uniting registers overall. Initialising the sparse-sets would take $O(|\mathbf{R}| \cdot |\mathbb{D}|)$ time, but $|\mathbb{D}|$ can be w.l.o.g. capped at $|w|$. $\square$

Note that also testing $w \in \mathcal{L}(\mathcal{A})$ with an infinite $\mathbb{D}$ with copy-free DRSA would be in $O(|\mathbf{R}| \cdot |w|)$.

7 Experimental Evaluation

We have implemented a prototype RSA-based regex matcher `rsamatch` [72] in Python and evaluated its performance against other state-of-the-art regex matchers on realistic ReDoS attack vectors involving backreferences. Our experiments are designed to evaluate the following hypotheses:

- (1) Existing regexes matchers are vulnerable when matching regexes with backreferences.
- (2) A large portion of regexes with backreferences contain only single-letter backreferences.
- (3) Most of the regexes with single-letter backreferences that are vulnerable to ReDoS can be converted to DRSA and matched with a predictable performance.

7.1 Experimental Setup

Regex Matchers and Data Sets. We compared `rsamatch` against a representative sample of state-of-the-art matchers, namely, `grep` [21], `pcre2` [32], Python RE [41] (denoted as `re`), standard library matchers of JavaScript [19], Java [50], and .NET [44] (denoted as `js`, `java`, and `.net` respectively). These are mostly backtracking-based, possibly in combination with other techniques, but they all mainly use their backtracking core when confronted with backreferences. We note that matchers for some scripting languages (Python’s `re` and `pcre2`) are actually implemented in C and not in the scripting language itself. Some very fast matchers such as RE2 [28] and especially HyperScan [73], based on automata and other techniques, are missing in the comparison since they do not support backreferences [27, 34]. All experiments were run on a Ubuntu GNU/Linux machine with the AMD EPYC 9124 CPU with 16 cores at 3.0 GHz and 94 GiB of memory (we note that the memory consumption of all matchers was far below the limits imposed by the hardware).

As our regexes, we started with 3,252 real-world regexes with backreferences obtained from the comprehensive data set *Lingua Franca* [17] and 8,794 regexes from the benchmarks of the ReDoS generator RENGAR [74], for a total of 12,046 regexes.⁷ From these, we picked regexes with single-letter backreferences because although multi-letter capture groups can be rewritten as single-letter ones (for a bounded length, cf. Section 6), the determinization often fails for them (Algorithm 1 returns $\perp$ due to overapproximation), which gave us 3,299 regexes (27 %).

ReDoS Inputs. In general, there are several flavours of ReDoS considered in the literature. In our evaluation, we consider the scenario where there is a regex (known to the attacker) running on a server (or a network intrusion detection system or a packet filter etc.) and the attacker provides an input that exhibits a costly matching, which is, to the best of our knowledge, the most realistic setting, since it has already led to several real-world ReDoSes [4, 22].⁸ The input of every experiment is a pair of a regex and an input string, which was created using the ReDoS generator RENGAR [74]. RENGAR was able to provide an attack vector for 1,335 regexes from the data set (40 %). Since in the considered ReDoS setting, one regex is used to match a large quantity of user inputs, we perform the determinization in `rsamatch` offline and then run the matching with the precomputed DRSA. Out of the 1,335 regexes with ReDoS inputs, our determinization algorithm returned $\perp$ on 43 regexes (3 %) and timed out (because of a regex complexity other than backreferences; e.g., most of the regexes start with an implicit “.*”, which already creates a nondeterministic choice at the beginning) on 41 of them (3 %). The average runtime of the determinization algorithm on successful

⁷In the *Lingua Franca* data set, 52 % of the regexes had capture groups of fixed lengths and 57 % had capture groups of bounded lengths (the fixed-length are a subset of the bounded-length). In the RENGAR data set, 22 % regexes had capture groups of fixed lengths and 24 % had capture groups of bounded lengths. In total, 30 % regexes had capture groups of fixed lengths and 33 % had capture groups of bounded lengths.

⁸Some of the other settings considered in the literature is, e.g., when the attacker can provide both the regex and the input. While such a situation may occur in the real-world, it is less common and much easier to exploit when backreferences are enabled due to the NP-hardness of their matching [2].

Table 1. A frequency table of run times and statistics for rsamatch and the other matchers on the 1,246 supported regexes with the RENGAR-generated inputs. The column **TOs** denotes timeout (> 100 s), column **> 1 s** is the sum of the values in all columns over 1 s, and the **errors** column shows the numbers of unsupported regexes. The numbers in the right-hand part of the table are for successful runs only and are in seconds.

tool	< 1 s	1–5 s	5–10 s	10–50 s	50–100 s	TOs	> 1 s	errors	mean	median	std. dev
rsamatch	1,244	2	0	0	0	0	2	—	0.299	0.16	0.303
grep	1,131	6	0	16	4	15	41	74	0.664	0.01	5.973
re	1,139	36	14	11	3	43	107	0	0.532	0.01	4.416
pcr2	1,001	14	4	4	0	3	42	203	0.144	0.01	1.219
js	660	8	1	11	1	33	54	532	0.531	0.06	3.702
java	595	18	5	7	5	19	54	597	0.996	0.05	6.438
.net	629	24	13	9	3	40	89	528	0.813	0.04	5.165

instances was 1.150 s, the median was 0.260 s, and the standard deviation was 3.064 s. Since the determinization needs to be performed only once for many runs of the matching algorithm, the cost of determinization can be amortized. RENGAR generated strings of the length between 55 and 90,124 characters, with the average of 34,521 characters. We set the timeout to quite generous 100 s.

Determinizability. In the previous paragraph, we analyzed on how many of the ReDoS-prone regexes we managed to construct a corresponding DRSA. Let us now discuss on how many of all input regexes (12,046) we managed to build the corresponding DRSA. From the 12,046 regexes, 2,944 contained unsupported syntax (we use Python frontend and its regex dialect), which gave us 9,102 regexes to start with. Out of these, our algorithm outputs DRSA on 3,138 of them (34.5 %). When we restrict ourselves to the single-letter ones, these were 3,434, making our algorithm successful on 91 % of them.

7.2 Results

In Table 1, we show a frequency table (the table form of a histogram) of run times of rsamatch and the other matchers on the 1,246 supported input regexes. We note that some of the regex matchers failed on a considerable number of the input regexes—this is due to different dialects of regexes used by the matchers and should not be considered their failure (rsamatch uses Python’s dialect, which is why re has 0 errors). The use of different dialects means that the semantics of one regex differs based on the matcher [17]; while this would be an issue if we were are cross-comparing the performance of the matchers, in this study, we are focusing on ReDoS vulnerability, where the concrete semantics of a regex is not so important.

The results show that rsamatch’s run time for the vast majority of benchmarks (all except two) is below one second (the run times for the two hardest benchmarks were 1.13 and 2.2 seconds). The other matchers, on the other hand, indeed struggled on these attack vectors considerably, often taking tens of seconds or even exceeding the 100 s timeout. Considering that the normal matching time on such strings is counted in milliseconds, they can indeed be considered manifestations of vulnerabilities, even for the relatively simple class of regexes with single-letter backreferences. If we fix a ReDoS threshold at 1 s, the column > 1 s counts the number of successful ReDoS attacks. Here, the second best matcher (after rsamatch) was pcre2 with 42 attacks (although it could not run on 203 benchmarks) and then grep with 41 attacks (with 74 benchmarks excluded due to an error).

In the right-hand part of the table, we give statistics about the run times of the matchers on the instances where they did not timeout (or fail): the arithmetic mean, the median, and the standard deviation. Although rsamatch is a prototype written in Python, its average time is comparable to the other matchers (also taking into account the fact that they timed out on a number of inputs). The median run time of rsamatch is higher than for the other matchers, but it should be significantly

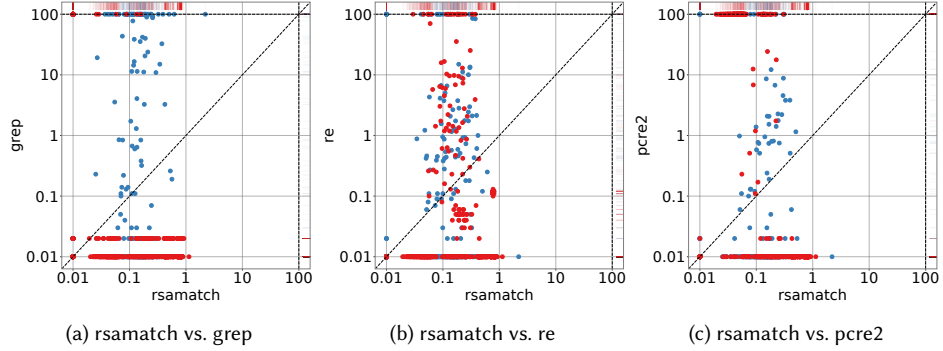


Fig. 6. Comparison of rsamatch with grep, re, and pcre2. Times are in seconds, axes are logarithmic. Dashed lines represent timeouts (100 s). Colours distinguish datasets: • Lingua Franca, • RENGAR.

decreased if reimplemented in a more efficient programming language, such as C/C++. The last column of the table also exhibits the robustness and predictability of the performance of rsamatch, since the standard deviation of the run times is (despite rsamatch’s prototype nature) an order of magnitude lower than for the other matchers.

We include a more detailed comparison with the best competing tools, grep, re, and pcre2, in the form of scatter plots shown in Fig. 6. One can clearly see the contrast in performance predictability, with our matcher almost always staying below the 1 s mark, on many being instances significantly faster than the competitors. Note that grep either managed to solve the matching problem almost instantly or timed out.

8 Related Work

Regex Matching. The literature on efficiently matching regexes is rich, with works ranging from matching feature-rich regexes in mainstream languages [7, 71], to instruction-level optimized regex matching in HyperScan [73], down to accelerating regex matching using FPGAs [11, 42]. Here we focus mainly on automata-based regex matching and regexes with backreferences.

Automata-based regex matching can be traced back to Thompson [66], who introduced a practical algorithm based on on-demand NFA determinization with caching. There have been several attempts of automata models for matching regexes with backreferences, but they have generally failed expectations of the community. Memory automata of Schmid [56] provide a natural extension of finite automata with a register bank, each register able to store a word, but do not allow determinization. The language-theoretic aspects of regexes with backreferences were further studied in [25]. Chida and Terauchi [13] show that lookaheads increase the expressive power of regexes with backreferences. Becchi and Crowley [8] encode backreferences into an extended model of NFAs using backtracking. Nogami and Terauchi [49] devise a quadratic-time (w.r.t. the input length) algorithm for matching a certain class of regexes with backreferences, which is incomparable to our class—they allow capture groups with unbounded length, but only one capture group and one reference to it; our work, on the other hand, allows multiple capture groups, multiple references to them, and our constructed automata often process the input in linear time (with a quadratic time worst-case guarantee). Namjoshi and Narlikar [47] extend Thompson’s algorithm [66] to a model similar to the models of [8, 56] in the obvious way by tracking the set of all possible configurations, which is inefficient (for each symbol, the number of operations proportional to the (potentially exponential) size of the set of configurations is required). Varatalu *et al.* [71] present

a derivative-based regex matcher supporting many extensions of classical regexes. They do not support backreferences, but it seems that our derivatives (cf. Section 6) could be combined with it to extend its usability. Moreover, there are also some proprietary *ad hoc* solutions that allow matching of regexes with backreferences [64], but they are mostly incomplete and with no guarantees.

In our work, we were inspired by *counting-set automata* introduced by Turoňová *et al.* [69], which use *sets of counter values* to compactly represent configurations of *counting automata* [33] (a restricted version of counter automata [45] with a bound on the value of counters for compact representation of finite automata), to obtain a deterministic model for efficient matching of regular expressions with repetitions.

ReDoS. While initially neglected, the potential of exploiting regex matching for denial of service was popularized by the successful attacks on StackOverflow [22] and Express.js [4]. While many industrial matchers handle basic regexes reasonably well [7, 21, 28, 71, 73], using extensions can make the matchers vulnerable to ReDoS; in [68], this was demonstrated even for highly optimized matchers such as RE2 [28] or HyperScan [73] (these two have backreferences explicitly disabled [27, 34]). Over the years, there have been many works dealing with the analysis of regexes for ReDoS vulnerability, sometimes including ReDoS generation, and proposals for how to avoid this attack vector (see, e.g., the works in [6, 16, 17, 29, 38, 40, 43, 51, 62, 63, 74–76] or the systematic literature review in [9]). On a more theoretical level, Terauchi [65] studies conditions for ReDoS vulnerability of regexes with backreferences and proposes a technique for transforming a class of memory automata to invulnerable regexes.

Automata Models for Data Words. The literature on automata over infinite alphabets is rich, see, e.g., the excellent survey by Segoufin [61] and the paper by Neven *et al.* [48]. Register automata were introduced (under the name *finite memory automata*) by Kaminski and Francez in [37] and their basic properties further studied by Sakamoto and Ikeda in [55]. Demri and Lazić study in [18] (non-)deterministic, universal, and alternating one-way and two-way register automata, and their relation to the linear temporal logic with the *freeze* quantifier, which can store the current data value into a register. In particular, they show that $LTL_1^f(X, U)$, i.e., the fragment of the logic with one freeze register and the *next* (X) and *until* (U) temporal operators captures the class of languages accepted by one-way alternating register automata with one register (ARA_1). Figueira [23] introduces alternating register automata with one register and two extra operations: *guess* and *spread* ($ARA_1(g, s)$). Intuitively, *guess* and *spread* can be used for existential and universal quantification over future and past data values respectively.

Set-augmented finite automata, introduced in [5], resemble RSAs in a way that they allow to store a set of values in a register, however, with noticeable restrictions: (i) set-augmented automata allow equality/disequality guard to be a singleton only, and (ii) the update function is restricted to storing the currently read data value to a single register (or no update is applied as a second option). It is not possible, e.g., to union the contents of two set-registers (which is a crucial operation in our determinization algorithm), or to empty a register. Due to these restrictions, RSAs have greater expression power compared to set-augmented automata (e.g., the parametric language defined in [5, Theorem 1] can be accepted by RSA_1 but not by a corresponding set-augmented automaton).

History-register automata, introduced in [70], are a similar model to RSA^m in a way that they also allow sets of values to be stored in registers. The key difference is the assignment of values to registers. The only way how to change a value in a register is add/remove the *currently processed* data value. It is not possible e.g., to union the content of two registers (which is, as noted above, crucial in our determinization), which is possible in RSA^m making our model more expressive. In particular,

in [30] we show that RSA^m are expressive at least as history-register automata and moreover, deterministic history-register automata are strictly less expressive than deterministic RSA^m .

9 Conclusion and Future Work

We have introduced register set automata, a class of automata over data words providing an underlying formal model for efficient matching of a subclass of regexes with backreferences. There are many challenges that we wish to address in the future: (i) improvement of our determinization algorithm to work on a larger class of input NRAs, (ii) explore other, more expressive, formal models that would allow deterministic automata models for a large class of regexes with backreferences (e.g., the hard regex in footnote² is beyond the power of DRSA), (iii) develop efficient toolbox for working with RSAs occurring in practice, and (iv) explore other approaches of how to efficiently deal with nondeterminism in practice, e.g., in a similar way as considered in [39] (which does not consider backreferences).

Data Availability Statement

An environment with the tools and data used for the experimental evaluation in the current study is available at [31].

Acknowledgments

We thank the anonymous reviewers for their feedback that improved the quality of the paper. This work was supported by the Czech Science Foundation projects 26-22640S and 25-17934S and the FIT BUT internal project FIT-S-26-9011. ■ This work has been executed under the project VASSAL: “Verification and Analysis for Safety and Security of Applications in Life” funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

References

- [1] Adar Weidman. 2025. Regular expression Denial of Service — ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. [Online; accessed 25-March-2025].
- [2] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 255–300.
- [3] Valentin M. Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theor. Comput. Sci.* 155, 2 (1996), 291–319. doi:10.1016/0304-3975(95)00182-4
- [4] Adam Baldwin. 2016. Regular Expression Denial of Service affecting Express.js. <https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43>. [Online; accessed 25-March-2025].
- [5] Ansuman Banerjee, Kingshuk Chatterjee, and Shibashis Guha. 2023. Set Augmented Finite Automata over Infinite Alphabets. In *Developments in Language Theory*. Springer Nature Switzerland, Cham, 36–50.
- [6] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting Input Sanitization for Regex Denial of Service. In *44th IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 883–895. doi:10.1145/3510003.3510047
- [7] Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1336–1360. doi:10.1145/3656431
- [8] Michela Becchi and Patrick Crowley. 2008. Extending finite automata to efficiently match Perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*. ACM Press, Madrid, Spain, 1–12. doi:10.1145/1544012.1544037
- [9] Masudul Hasan Masud Bhuiyan, Berk Çakar, Ethan H. Burmane, James C. Davis, and Cristian-Alexandru Staicu. 2025. SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS). In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2025, Hanoi, Vietnam, August 25-29, 2025*. ACM, 1659–1675. doi:10.1145/3708821.3733912
- [10] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. 2013. A Fresh Approach to Learning Register Automata. In *Developments in Language Theory (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 118–130. doi:10.1007/978-3-642-38771-5_12

- [11] Milan Ceska, Vojtech Havlena, Lukáš Holík, Jan Korenek, Ondrej Lengál, Denis Matousek, Jirí Matousek, Jakub Semric, and Tomáš Vojnar. 2019. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 109–117. doi:10.1109/FCCM.2019.00025
- [12] Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2020. A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving. In *Programming Languages and Systems*, Vol. 12470. Springer International Publishing, Cham, 343–363. doi:10.1007/978-3-030-64437-6_18 Series Title: Lecture Notes in Computer Science.
- [13] Nariyoshi Chida and Tachio Terauchi. 2023. On Lookaheads in Regular Expressions with Backreferences. *IEICE Trans. Inf. Syst.* 106, 5 (2023), 959–975. doi:10.1587/TRANSINF.2022EDP7098
- [14] Loris D’Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20–21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 541–554. doi:10.1145/2535838.2535849
- [15] James C. Davis. 2019. Rethinking Regex engines to address ReDoS. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1256–1258. doi:10.1145/3338906.3342509
- [16] James C. Davis. 2020. *On the Impact and Defeat of Regular Expression Denial of Service*. Ph.D. Dissertation. Virginia Tech, Blacksburg, VA, USA. <http://hdl.handle.net/10919/98593>
- [17] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren’t regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 443–454. doi:10.1145/3338906.3338909
- [18] Stéphane Demri and Ranko Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic* 10, 3 (April 2009), 1–30. doi:10.1145/1507244.1507246
- [19] V8 JavaScript Engine. 2024. V8 regular expression source code. Online. Available: <https://github.com/v8/v8/tree/main/src/regexp>, [Accessed: 2024-11-17].
- [20] Javier Esparza and Michael Blondin. 2023. *Automata Theory: An Algorithmic Approach*. MIT Press.
- [21] M. Haertel et al. 2022. GNU grep. Online. Available: <https://www.gnu.org/software/grep/>, [Accessed: 2024-11-17].
- [22] Stack Exchange. 2016. Outage Postmortem. <http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016>.
- [23] Diego Figueira. 2012. Alternating register automata on finite words and trees. *Logical Methods in Computer Science* 8, 1 (March 2012), 22. doi:10.2168/LMCS-8(1:22)2012
- [24] Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory Comput. Syst.* 53, 2 (2013), 159–193. doi:10.1007/S00224-012-9389-0
- [25] Dominik D. Freydenberger and Markus L. Schmid. 2019. Deterministic regular expressions with back-references. *J. Comput. System Sci.* 105 (2019), 1–39. doi:10.1016/j.jcss.2019.04.001
- [26] Bharat Garhwal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. 2020. Grey-Box Learning of Register Automata. In *Integrated Formal Methods (Lecture Notes in Computer Science)*, Brijesh Dongol and Elena Troubitsyna (Eds.). Springer International Publishing, Cham, 22–40. doi:10.1007/978-3-030-63461-2_2
- [27] Google. 2022. *RE2 Issue Tracker: Feature request #101: Add backreference support*. <https://github.com/google/re2/issues/101>
- [28] Google. 2025. *RE2*. <https://github.com/google/re2>
- [29] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. 2023. Improving Developers’ Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21–25, 2023*. IEEE, 1238–1255. doi:10.1109/SP46215.2023.10179442
- [30] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Jan Vašák, and Sabína Gulčíková. 2026. Towards Efficient Matching of Regexes with Backreferences using Register Set Automata (Technical Report). *CoRR* 2205.12114 (2026). arXiv:2205.12114 <https://arxiv.org/abs/2205.12114>
- [31] Vojtěch Havlena, Lukáš Holík, Jan Vašák Ondřej Lengál, and Sabína Gulčíková. 2026. *Artifact for the PLDI’26 paper “Towards Efficient Matching of Regexes with Backreferences using Register Set Automata”*. doi:10.5281/zenodo.19223981
- [32] P. Hazel. 2024. Perl-compatible Regular Expressions. Online. Available: <https://www.pcre.org/>, [Accessed: 2024-11-17].
- [33] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2019. Succinct Determinisation of Counting Automata via Sphere Construction. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 468–489. doi:10.1007/978-3-030-34175-6_24

- [34] Intel. 2022. *HyperScan manual, unsupported features*. <https://intel.github.io/hyperscan/dev-reference/compilation.html#unsupported-constructs>
- [35] Radu Iosif and Xiao Xu. 2019. Alternating Automata Modulo First Order Theories. In *Computer Aided Verification (Lecture Notes in Computer Science, Vol. 11562)*. Springer International Publishing, Cham, 43–63. doi:10.1007/978-3-030-25543-5_3
- [36] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. Learning register automata: from languages to program structures. *Machine Learning* 96, 1 (July 2014), 65–98. doi:10.1007/s10994-013-5419-7
- [37] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (Nov. 1994), 329–363. doi:10.1016/0304-3975(94)90242-9
- [38] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7873)*, Javier López, Xinyi Huang, and Ravi S. Sandhu (Eds.). Springer, 135–148. doi:10.1007/978-3-642-38631-2_11
- [39] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-hardware codesign for efficient in-memory regular pattern matching. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 733–748. doi:10.1145/3519939.3523456
- [40] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Reveal: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1468–1484. doi:10.1109/SP40001.2021.00062
- [41] F. Lundh and A. M. Kuchling. 2023. Python Standard Library: re module. Online. Available: <https://docs.python.org/3/library/re.html>, [Accessed: 2024-11-17].
- [42] Denis Matousek, Jiri Matousek, and Jan Korenek. 2018. High-Speed Regular Expression Matching with Pipelined Memory-Based Automata. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 214. doi:10.1109/FCCM.2018.00048
- [43] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. 2022. Regulator: Dynamic Analysis to Detect ReDoS. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4219–4235. <https://www.usenix.org/conference/usenixsecurity22/presentation/mclaughlin>
- [44] Microsoft. 2024. .NET 8.0.110 regular expressions in System.Text.RegularExpressions. Online. Available: <https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions?view=net-8.0>, [Accessed: 2024-11-17].
- [45] Marvin L. Minsky. 1961. Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines. *Annals of Mathematics* 74, 3 (1961), 437–455. doi:10.2307/1970290
- [46] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. 2018. Polynomial-Time Equivalence Testing for Deterministic Fresh-Register Automata. (2018), 14 pages. doi:10.4230/LIPICS.MFCS.2018.72
- [47] Kedar S. Namjoshi and Girija J. Narlikar. 2010. Robust and Fast Pattern Matching for Intrusion Detection. In *INFOCOM'10*. IEEE, 740–748. doi:10.1109/INFCOM.2010.5462149
- [48] Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435. doi:10.1145/1013560.1013562
- [49] Taisei Nogami and Tachio Terauchi. 2025. Efficient Matching of Some Fundamental Regular Expressions with Backreferences. In *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025, August 25-29, 2025, Warsaw, Poland (LIPIcs, Vol. 345)*, Pawel Gawrychowski, Filip Mazowiecki, and Michal Skrzypczak (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 81:1–81:19. doi:10.4230/LIPICS.MFCS.2025.81
- [50] OpenJDK. 2024. JDK 17 API Documentation: Regular Expressions. Online. Available: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/package-summary.html>, [Accessed: 2024-11-17].
- [51] Francesco Parolini and Antoine Miné. 2022. Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks. In *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8-10, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13299)*, Yamine Ait Ameur and Florin Craciun (Eds.). Springer, 73–91. doi:10.1007/978-3-031-10363-6_6
- [52] Michael O. Rabin and Dana S. Scott. 1959. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 3, 2 (1959), 114–125. doi:10.1147/rd.32.0114
- [53] Regex101.com. 2025. <https://regex101.com>. [Online; accessed 25-March-2025].
- [54] M. Roesch et al. 2022. *Snort: A Network Intrusion Detection and Prevention System*. Cisco. <http://www.snort.org>
- [55] Hiroshi Sakamoto and Daisuke Ikeda. 2000. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 231, 2 (2000), 297–308. doi:10.1016/S0304-3975(99)00105-X

- [56] Markus L. Schmid. 2016. Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing. 249 (Aug. 2016), 1–17. doi:10.1016/j.ic.2016.02.003
- [57] Sylvain Schmitz. 2016. Complexity Hierarchies Beyond Elementary. *ACM Transactions on Computation Theory* 8, 1 (Feb. 2016), 1–36. doi:10.1145/2858784
- [58] Sylvain Schmitz. 2017. *Algorithmic Complexity of Well-Quasi-Orders*. Habilitation à diriger des recherches. École normale supérieure Paris-Saclay. <https://tel.archives-ouvertes.fr/tel-01663266>
- [59] Sylvain Schmitz and Philippe Schnoebelen. 2012. Algorithmic Aspects of WQO Theory. (Aug. 2012). <https://cel.archives-ouvertes.fr/cel-00727025>
- [60] Sylvain Schmitz and Philippe Schnoebelen. 2013. The Power of Well-Structured Systems. In *CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8052)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.). Springer, 5–24. doi:10.1007/978-3-642-40184-8_2
- [61] Luc Segoufin. 2006. Automata and Logics for Words and Trees over an Infinite Alphabet. In *Computer Science Logic*. Vol. 4207. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–57. doi:10.1007/11874683_3
- [62] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 225–235. doi:10.1145/3238147.3238159
- [63] Weihao Su, Hong Huang, Rongchen Li, Haiming Chen, and Tingjian Ge. 2024. Towards an Effective Method of ReDoS Detection for Non-backtracking Engines. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. <https://www.usenix.org/conference/usenixsecurity24/presentation/su-weihao>
- [64] Nvidia Mellanox team. 2021. Personal communication.
- [65] Tachio Terauchi. 2025. On DoS Vulnerability of Regular Expressions, with and Without Backreferences. In *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*. IEEE, 190–204. doi:10.1109/CSF64896.2025.00011
- [66] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422. doi:10.1145/363347.363387
- [67] Milo Tova, Suciu Dan, and Vianu Victor. 2000. Typechecking for XML transformers. In *POD'00*. ACM, Dallas, USA, 11–22. doi:10.1145/335168.335171
- [68] Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2022. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4165–4182. <https://www.usenix.org/conference/usenixsecurity22/presentation/turonova>
- [69] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 218:1–218:30. doi:10.1145/3428286
- [70] Nikos Tzevelekos and Radu Grigore. 2013. History-Register Automata. In *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–33.
- [71] Ian Erik Varatalu, Margus Veanes, and Juhan P. Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL (2025), 1–32. doi:10.1145/3704837
- [72] Jan Vašák. 2026. rsamatch. <https://github.com/VeriFIT/RegisterSetAutomata>
- [73] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *NSDI'19*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 631–648. <https://www.usenix.org/conference/nsdi19/presentation/wang-xiang>
- [74] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, and Wei Huo. 2023. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2427–2443. doi:10.1109/SP46215.2023.10179328
- [75] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. 2016. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9705)*, Yo-Sub Han and Kai Salomaa (Eds.). Springer, 322–334. doi:10.1007/978-3-319-40946-7_27
- [76] Valentin Wüstholtz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). 3–20. doi:10.1007/978-3-662-54580-5_1

Received 2025-11-14; accepted 2026-04-03