BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REDUCING SIZE OF NONDETERMINISTIC AUTOMATA WITH SAT SOLVERS

REDUKCE VELIKOSTI KONEČNÝCH AUTOMATŮ POMOCÍ SAT SOLVERU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL ŠEDÝ**
AUTOR PRÁCE

SUPERVISOR **Mgr. LUKÁŠ HOLÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Department of Intelligent Systems (DITS) Academic year 2020/2021

Bachelor's Thesis Specification

23436

Student:	**Šedý Michal**
Programme:	Information Technology
Title:	**Reducing Size of Nondeterministic Automata with SAT Solvers**
Category:	Algorithms and Data Structures

Assignment:
In this work, we will attempt to develop new methods for reducing size of non-deterministic finite automata that go beyond existing techniques based on merging simulation equivalent states.

1. Study the principles of simulation based size reduction of automata, familiarize yourself with SAT/SMT solving.
2. Investigate new means of reducing size of automata. Start from optimizing simple "confluent" parts of the automaton (sub-graphs with single source and single target state and several states in the middle) based on optimization of the sum of products, preferably using SAT/SMT solvers.
3. Based on your research, propose a new technique of automata size reduction.
4. Implement the proposed technique and evaluate its reduction capabilities respective to known reduction techniques.

Recommended literature:

- Lorenzo Clemente, Richard Mayr:Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. 15(1) (2019)
- Ilie L., Navarro G., Yu S. (2004) On NFA Reductions. In: Karhumäki J., Maurer H., Păun G., Rozenberg G. (eds) Theory Is Forever. Lecture Notes in Computer Science, vol 3113. Springer, Berlin, Heidelberg

Requirements for the first semester:

- Item 1 of the assignment,
- part of the work on 2,
- at least a small part of the text of the work.

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

Supervisor:	**Holík Lukáš, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 12, 2021
Approval date:	April 7, 2021

Abstract

Nondeterministic finite automata (NFA) are widely used in computer science fields, such as regular languages in formal language theory, high-speed network monitoring, image recognition, hardware modeling, or even in bioinformatic for the detection of the sequence of nucleotide acids in DNA. They are also used in regular mode checking, in string solving, in verification of pointer manipulating programs, for construction of linear arithmetic equations and inequalities, for decision in WS1S and WS2S logic, and many others. Automata minimization is a fundamental technique that helps to decrease resource claims (memory, time, or a number of hardware components) of implemented automata and speed up automata operations. Commonly used minimization techniques, such as state merging, transition pruning, and saturation, can leave potentially minimizable automaton subgraphs with duplicit language information. These fragments consist of a group of states, where the part of language of one state is piecewise covered by the other states in this group. The thesis describes a new minimization approach, which uses SAT solver, which provides information for efficient minimization of these so far nonminimizable automaton parts. Moreover, the newly investigated method, which only uses solver information and state merging, can minimize the automaton similarly and on automata with low transition count faster than a tool RABIT/Reduce, which uses state merging and transition pruning.

Abstrakt

Nedeterministické konečné automaty (NKA) jsou široce využívány v počítačové vědě, například v oblasti formálních jazyků pro reprezentaci regulárních jazyků, k monitorování vysokorychlostních sítí, rozpoznávání obrazu, modelování hardware, nebo dokonce v bioinformatice pro vyhledávání sekvencí nukleotidových kyselin v DNA. NKA jsou také používány v abstraktním regulárním model checkingu, dále ve verifikaci programů manupulujících s řetězci, ve verifikaci programů využívajících ukazatele, pro konstrukci lineárních rovnic a nerovnic, v rozhodovacích procedurách WS1S a WS2S logiky a mnohých dalších. Minimalizace automatů je základní technikou, která pomáhá snižovat nároky na zdroje (paměť, čas nebo množství hardwarových komponentů) a urychlovat operace prováděné na automatech. Běžně používané minimalizační techniky, jakými jsou slučování stavů, odstraňování hran přechodů nebo saturace, mohou v automatech zanechat potenciální minimalizovatelné podgrafy obsahující duplicitní jazykovou informaci. Tyto fragmenty sestávají ze skupiny stavů, kde je již část jazyka jednoho stavu pokryta jazyky ostatních stavů z této skupiny. Tato práce popisuje novou techniku využívající SAT solver, který poskytuje informaci umožňující minimalizovat tyto doposud neminimalizovatelné části automatů. Nově vyvíjená metoda, která využívá pouze informaci od SAT solveru a slučování stavů minimalizuje automaty podobně efektivně, a v případě automatů s nízkým počtem přechodů dokonce rychleji než nástroj RABIT/Reduce, který využívá slučování stavů a odstraňování hran.

Keywords

finite automata, nondeterministic finite automata, minimization, reduction, SAT solver, Z3 solver, state equivalency, language equivalency, state merging, quotienting

Klíčová slova

konečné automaty, nedeterministické konečné automaty, minimalizace, redukce, SAT solver, Z3 solver, ekvivalence stavů, ekvivalence jazyků, slučování stavů, quotienting

Reference

ŠEDÝ, Michal. *Reducing Size of Nondeterministic Automata with SAT Solvers.* Brno, 2021. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

Nedeterministické konečné automaty (NKA) prezentoval Michael Rabin a Dana Scott v [28]. Ve srovnání s deterministickými konečnými automaty (DKA) se vyznačují schopností přechodu do více následujících stavů na základě stejného přijatého znaku. Díky této vlastnosti umožňují NKA reprezentovat jazyk za pomoci menšího množství stavů a přechodů než jeho deterministická varianta. Nicméně každá mince má dvě strany. V důsledku nedeterminismu je minimalizace NKA mnohem obtížnější. Tato práce popisuje novou minimalizační techniku, která umožňuje redukovat velikost doposud neminimalizovatelných částí automatů. Předmětem práce není hledání minimální formy NKA, ale pouze redukce jeho velikosti. Pojmem minimalizace budeme v této práci rozumět pouze redukci velikosti automatů.

Nedeterministické automaty jsou používány k reprezentaci regulárních jazyků, v aplikacích pro validaci dat, internetových vyhledávačích, rozpoznávání řetězců, monitorování síťového provozu, dokonce i v bioinformatice pro vyhledávání sekvencí nukleových kyselin v DNA [3], a mnoha dalších. Příkladem využití NKA je detekce řetězců v síťovém provozu. V důsledku stále se zvyšujícího objemu dat přenášených po síti a také rychlostí přenosu je potřeba zvyšovat také rychlost jejich skenování. Standardní softwarová řešení pro vyhledávání řetězců, kterými mohou být jak zajímavé statistické údaje, tak škodlivý kód, již nejsou při vysokých rychlostech přenosu použitelné. Pro rychlosti nad 100 Gbps je potřeba vytvořit hardwarovou implementaci vyhledávání řetězců [32, 25]. Pomocí technik [30] je možné reprezentovat NKA přímo na FPGA. Z důvodu ušetření místa, prostředků a nízké ceny vyráběných komponentů je vhodné automat minimalizovat. Nedeterministické konečné automaty jsou také používány pro verifikaci programů [19], v abstraktním regulárním model chackingu [9], dále ve verifikaci programů manupulujících s řetězci [2], při verifikaci programů využívajících ukazatele [18], pro konstrukci lineárních rovnic a nerovnic [33], v rozhodovacích procedurách WS1S [16, 15] a WS2S [15] logiky a mnohých dalších. Pro zvýšení rychlosti operací prováděných nad automaty je vhodné redukovat jejich velikost.

V současnosti existuje mnoho efektivních minimalizačních technik. Nejznámější minimalizační metodou je slučování ekvivalentních stavů [6, 10, 24, 22], které slučuje dva jazykově ekvivalentní stavy do jednoho. Dalšími úspěšnými postupy v oblasti minalizace jsou odstraňování hran přechodů [10, 13] a jejich přidávání (saturace) [6, 13]. Hrana přechodu může být odstraněna, když již existuje vhodnější hrana (existuje stav se silnějším, nebo ekvivalentním jazykem). Ve srovnání s odstraňování přechodů, saturace přechody do automatu přidává. Přidání nové hrany může umožnit další slučování stavů nebo odstraňování hran. Přechod může být přidán pouze tehdy, existuje-li již silnější, nebo stejný přechod. Bez ohledu na to, jak jsou stávající minimalizační techniky robustní, nejsou všemocné. Stále zanechávají v automatech potenciálně minimalizovatelné podgrafy. Tyto fragmenty sestávají ze skupiny stavů, kde je část jazyka jednoho z nich po pokryt jazyky ostatních stavů. Tento stav představuje v automatu duplicitní informaci, ale nemusí být detekován stávajícími metodami, protože mezi jazyky stavů nemusí existovat inkluze.

Tato práce vyvíjí metodu, která dokáže minimalizovat tyto doposud neminimalizovatelné části automatu. Nový postup pracuje se skupinami stavů, které mají společného předchůdce, nebo následníka. Všechny tyto stavy jsou nahrazeny novými stavy, které obsahují maximálně jednu vstupní a maximálně jednu výstupní hranu přechodu. Nad takovými stavy je za pomocí SAT solveru provedeno nejoptimálnější slučování stavů. Solver aproximuje silně (v případě automatů s nízkým počtem přechodů také rychleji) výsledky

minimalizace provedené nástrojem RABIT/Reduce[1], který využívá slučování a odstraňování hran přechodu.

Obsah kapitol

Kapitola 2 je věnována základním teoretickým pojmům týkajících se nedeterministických konečnách automatů. Nejprve je definován samotný NKA, poté související pojmy, jakými jsou konfigurace, přechod, jazyky automatu a jazyky stavu. V kapitole 3 jsou definovány existující minimalizační techniky využívané nástrojem RABIT/Reduce (slučování stavů, odstraň-ování hran přechodů a saturace). Kapitola 4 obsahuje základní myšlenky práce založené na chování slučování stavů. Kapitola 5 popisuje kódování problému minimalizace automatu, které se skládá z definice proměnných, slučovací formule a pravidel. V kapitole 6 jsou uvedeny výsledky experimentů minimalizace s využitím SAT solveru a také porovnání výsledků s existujícím nástrojem RABIT/Reduce.

[1] nástroj RABIT/Reduce je dostupný na `http://languageinclusion.org/doku.php?id=tools`.

Reducing Size of Nondeterministic Automata with SAT Solvers

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Mgr. Lukáš Holík, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michal Šedý
May 7, 2021

Acknowledgements

I would like to thank my supervisor Mgr. Lukáš Holík, Ph.D. for his help.

Contents

Chapter 1

Introduction

Nondeterministic finite automata (NFA) were investigated by Michael Rabin and Dana Scott [28]. In comparison with deterministic finite automata (DFA), NFA can choose from more than one transition after receiving the letter. This feature allows NFA to represent the language with fewer states and transitions than its deterministic variant. However, there are two sides to every story. The NFA is much harder to minimize. This work investigates a new approach for minimizing of so far nonminimizable parts of automata. A goal of the investigated approach is not to find the minimal form of NFA, but only to reduce automata size. In this thesis, we will use the term minimization just for the reduction of automata size.

Nondeterministic finite automata are often used for a representation of regular languages, in data validation, web searching engines, pattern recognition, in network traffic monitoring, even in genetic for matching of the sequence of nucleotide acids on DNA [3], and many others. An example of NFA usage is a representation of the regular expression for pattern matching in network traffic. Due to an increasing amount of data transmitted over the network and so increasing speed, it is necessary to improve the data scanning speed. Standard software solutions that can detect data fragments cannot be used in high-speed networks. For the speed over 100 Gbps, it is required to implement a hardware analyzer [32, 25]. The NFA can be implemented right on FPGA using the technique [30]. To save space, resources, and cost of manufactured components, it is advisable to minimize the original automaton. NFAs are also used in program verification [19], in abstract regular model checking [9], in verification of programs using pointers [18], in string solving [2], for construction of linear arithmetic equation and inequalities [33], for decision procedures in WS1S [16, 15] and WS2S [15] logic, and many others. It is necessary to reduce the size of automata to speed up the operations performed on them.

Nowadays, many efficient minimization techniques exist. The well-known method, state merging [6, 10, 24, 22], merges two language equivalent states into one. Other successful procedures in the field of minimization are transition pruning [10, 13] and transition adding (saturation) [6, 13]. The transition can be pruned if the better transition already exists (a state with stronger or equal language exists). On the contrary, saturation adds new already existing transitions. This addition can help to continue in state merging or transition pruning. Despite the high efficiency of these methods, they are not omnipotent. They can still leave potentially minimizable automata subgraphs. These fragments consist of a set of states, where part of the language of some state is piecewise covered by other states. Each state can have a unique language, so language inclusion, which is necessary for minimization, does not exist.

The work investigates a method for minimizing this so far unsolvable automata subgraphs. The method works with sets of states with common successors or ancestors. All states from a set are substituted by states with the maximal one incoming and outcoming transition, then SAT solver is used for maximizing the number of merged states. In comparison to an existing tool RABIT/Reduce[1] which uses state merging and transition pruning, the solver gives a strong, and on automata with small count of transitions faster, approximation of RABIT minimization.

Plan of the thesis

Chapter 2 is dedicated to the theoretical background of nondeterministic finite automata. First, the NFA will be defined and the related terms such as configuration, transition, and languages of the automaton and of the state. The existing minimization techniques such as state merge, transition pruning, and saturation, which the compared tool RABIT/Reduce uses, are listed in Chapter 3. After introducing minimization approaches, the basic thoughts, based on the behavior of state merge, are presented in Chapter 4. Chapter 5 is devoted to the coding of a minimization problem with SAT solver. The creation of solver variables, merge formula, and rules is described here. The limitation of the investigated SAT-solver-based minimization approach is shown at the end of the chapter. Chapter 6 displays a comparison of the experiments of automata minimization performed with a SAT-solver-based approach and the tool RABIT/Reduce.

[1] RABIT is available at `http://languageinclusion.org/doku.php?id=tools`.

Chapter 2

Preliminaries

Basic concepts of nondeterministic finite automata (NFA) and its notations, which are required for an easier understanding of the following thesis, are presented in this chapter. The formal definition of NFA (taken from [31, p.53]) and related concepts, such as automata configuration, transition, ancestors, successors, reachable or dead states are established in Section 2.1. The definitions of automata configuration and transition are taken from [26]. An elementary property of an automaton is a language, which defines accepting strings. These definitions are presented in Section 2.2. The language is not calculated only over the machine. Section 2.3 brings definitions for the language of state, such as forward language, backward language, and languages defined specially for this work. The last Section 2.4 of this chapter is dedicated to simulation (the definition is taken from [1]). The simulation is an approximation method widely used in a calculation of language relations, due to its higher speed in comparison with accurate methods.

2.1 The Formal Definition of a Nondeterministic Finite Automaton

Nondeterministic finite automata are a generalization of deterministic automata. Both contain states, transitions between them, and an alphabet of characters that the automaton read. In comparison to deterministic automata, nondeterministic automata have a set of initial and final states, not only one initial and one final state. Initial states represent the entry points of the automaton. Reading of an evaluated string, consisting of the characters of the machine alphabet, always starts in these states. Final states represent the place, which marks the input string as accepted. The string is accepted only if the automaton is in at least one of these final states, after reading the entire string. States are interconnected by transitions with alphabet symbols. Together they form an oriented graph. Based on the read symbol, a corresponding transition with the same symbol will be made. Compared to deterministic automata, where exactly one transition can be made based on one character, nondeterministic automata, as the name suggests, can choose from several transitions. If the nondeterministic transition is reached, then the next evaluation is split and the string is accepted if at least one of the splitted evaluations has ended in the final state.

To write the formal definition, we need to set up some additional notations. For a set Q, we write $P(Q)$ to be the collection of all subsets of Q. Here $P(Q)$ is called the *power* *set* of Q. For any alphabet Σ, we write Σ_ϵ to be $\Sigma \cup \{\epsilon\}$. Now we can write the formal description of the type of the transition function in an NFA $\delta : Q \times \Sigma_\epsilon \longrightarrow P(Q)$.

Definition 2.1 ***Nondeterministic Finite Automaton*** *is a 5-tuple* $M = (Q, \Sigma, \delta, I, F)$

1. Q *is a finite set of states,*

2. Σ *is an alphabet,*

3. $\delta : Q \times \Sigma_\epsilon \longrightarrow P(Q)$ *is a transition function,*

4. $I \subseteq Q$ *is a finite set of initial states, and*

5. $F \subseteq Q$ *is a finite set of final states.*

We will use $p \xrightarrow{a} q$ to denote the transition $((p, a), q)$, where $p, q \in Q$ and $a \in \Sigma$. The transition $p \xrightarrow{a} q$ says that after the reading of a character a in the state p the transition from state p to q will be made.

Definition 2.2 *Let* $M = (Q, \Sigma, \delta, I, F)$ *be the NFA,* $s \in Q$*, and* $a \in \Sigma$*. Then the* ***reverse transition function*** δ^{-1} *is defined as* $\delta^{-1}(s, a) = \{q \,|\, \forall\, q \in Q$ *where* $s \in \delta(q, a)\}$*.*

Definition 2.3 *Let* $M = (Q, \Sigma, \delta, I, F)$ *be the NFA and* $s \in Q$*. Then the* ***ancestors*** *of a state* s *is defined as* $anc(s) = \{q \,|\, q \in Q,\ \forall\, a \in \Sigma,\ q \in \delta^{-1}(s, a)\}$*.*

Definition 2.4 *Let* $M = (Q, \Sigma, \delta, I, F)$ *be the NFA and* $s \in Q$*. Then the* ***successors*** *of a state* s *is defined as* $succ(s) = \{q \,|\, q \in Q,\ \forall\, a \in \Sigma,\ q \in \delta(s, a)\}$*.*

Definition 2.5 *Let* $M = (Q, \Sigma, \delta, I, F)$ *be the NFA. Then the* ***configuration*** *is string* $\chi \in Q\Sigma^*$*.*

The automaton configuration displays information about the current state (not all states of the splitted evaluation, but only the focused one) and the remaining string at the input. For example, if the automaton M is in the state q and the string ab remains at the input, then the configuration takes the form qab. The attentive reader will certainly notice that the exact state of the automaton can be described as a set of configurations.

Definition 2.6 *Let* $M = (Q, \Sigma, \delta, I, F)$ *be the NFA, and* paw*,* qw *be two configurations of* M*, where* $p, q \in Q$*,* $a \in \Sigma_\epsilon$*, and* $w \in \Sigma^*$*. Let* $r : p \xrightarrow{a} q \in \delta$*. Then* M *can make* ***transition*** *from* qaw *to* qw *using* r*, write as* $paw \vdash qw\ [r]$ *or simply* $paw \vdash qw$*.*

Let χ be configuration. M makes zero transitions from χ to χ. Write: $\chi \vdash^0 \chi\ [\epsilon]$ or simply $\chi \vdash^0 \chi$.

Let $\chi_0, \chi_1 \ldots \chi_n$ be sequence of configurations for $n \leq 1$ and $\chi_{i-1} \vdash \chi_i\ [r_i]$, where $r_i \in \delta$ for all $i = 1 \ldots n$, what means: $\chi_0 \vdash \chi_1\ [r_1] \cdots \vdash \chi_n\ [r_n]$. Then M makes n transitions from χ_0 to χ_n. Write: $\chi_0 \vdash^n \chi_n\ [r_1 \ldots r_n]$ or simply $\chi_0 \vdash^n \chi_n$.
If $\chi_0 \vdash^n \chi_n\ [\rho]$ for some $n \geq 1$, then we write $\chi \vdash^+ \chi_n\ [\rho]$ or simply $\chi \vdash^+ \chi_n$.
If $\chi_0 \vdash^n \chi_n\ [\rho]$ for some $n \geq 0$, then we write $\chi \vdash^* \chi_n\ [\rho]$ or simply $\chi \vdash^* \chi_n$.

Already during the creation of the automaton, but rather during its modifications, states, whose presence or absence does not affect the language of the automaton, can arise. There are two types of *useless* states, these elementary types can be combined.

The first type is a state that can have successors, and even paths to final states can exist, but no path leads to this state from the initial state. This state is called an *unreachable* state. Unreachability may be caused by the lack of state ancestor or affiliation in the unreachable subgraph.

Definition 2.7 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, then the state $q \in Q$ is* ***reachable*** *if exists $w \in \Sigma^*$ for which $q_0 w \vdash^* q$, where $q_0 \in I$. Otherwise, it is unreachable.*

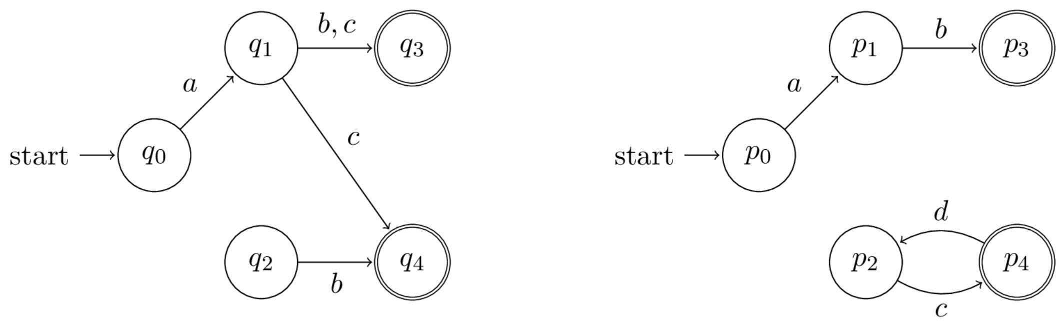


Figure 2.8: Two automata with unreachable states. For the left automaton, it is state q_2 and for the right automaton, it is state p_2 and p_3.

The second type of useless state is the so-called *dead* state. This is the case where no path from a state to the final states exists, regardless of the possible paths from the initial state. The dead nature of the state may be caused by the nonexistence of the successor, or its closure in the nonterminating part of the automaton subgraph, for example, in an isolated loop.

Definition 2.9 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then state $q \in Q$ is* ***undead*** *if exists $w \in \Sigma^*$ for which $qw \vdash^* f$, where $f \in F$. Otherwise, it is dead.*

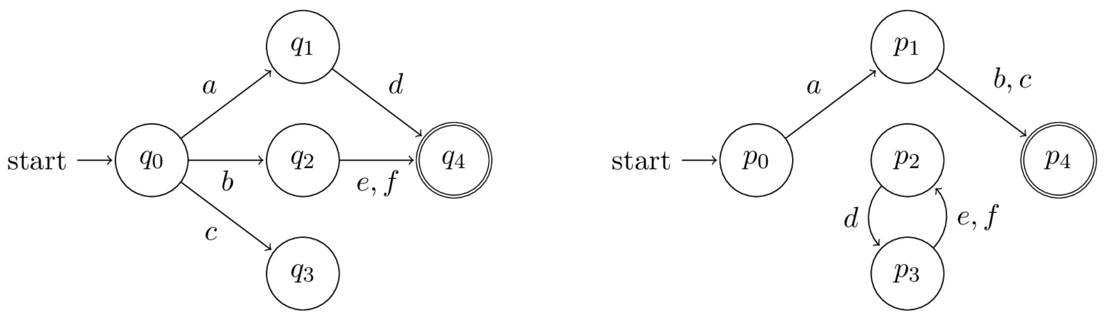


Figure 2.10: Two automata with dead states. For the left automaton, it is state q_3 and for the right automaton, it is state p_2 and p_3.

Both types of useless states can be eliminated. They can be easily removed because both unavailable and dead states do not belong to any part of the automaton that defines the language. Elimination of useless states is the most primitive way of minimization. All states that belongs to the set $\{Q \setminus (\mathit{undead} \cap \mathit{reachable})\}$ are useless ant therefore can be removed.

2.2 NFA Languages

As already mentioned, one of the main features of nondeterministic finite automata is their language. A language is a set of accepting strings defined by the automaton topology. For example, the language over the alphabet $\Sigma = \{0, 1\}$ might define strings with a maximal length of 4 and an even count of 0. The following definition of an acceptable string is taken from [31, p. 54].

Definition 2.11 *Let $M = (Q, \Sigma, \delta, I, F)$ be an NFA and w a string over the alphabet Σ. Then we say that w is* ***accepting string*** *in the form $w = y_1 y_2 \dots y_n$, where $y_i \in \Sigma$ for $i = 1 \dots n$ if exists such a sequence of states $r_0 r_1 \dots r_n \in Q$, with three conditions:*

1. $r_0 \in I$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, *for* $i = 0 \dots n-1$, *and*
3. $r_n \in F$

Condition 1 says that the reading starts out in the initial state. Condition 2 says that state r_{i+1} is one of the allowable next states when M is in state r_i after reading y_{i+1}. Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states, and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is a final state.

Definition 2.12 *Let $M = (Q, \Sigma, \delta, I, F)$ be an NFA. Then* ***accepting language*** *is defined as follows: $L(M) = \{w \mid w \in \Sigma^*,\ q_0 w \vdash^* f,\ \text{where } q_0 \in I \text{ and } f \in F\}$.*

The accepting language of NFA is the set of accepting strings. Where at least one final state is reached after reading all its input characters.

Definition 2.13 *Let $M = (Q_M, \Sigma_M, \delta_M, I_M, F_M)$ and $N = (Q_N, \Sigma_N, \delta_N, I_N, F_N)$ be two NFAs. We say that automata M and N are* ***equivalent*** *only if $L(M) \equiv L(N)$.*

According to definition 2.14, two automata M and N are equivalent if for each string $w_M \in L(M)$ exists such a sequence of transitions $w_M p_0 \vdash^* f_N$ in N, where $p_0 \in I_N$, $f_N \in F_N$, and for each $w_N \in L(N)$ exists such a sequence of transitions $w_N q_0 \vdash^* f_M$ in M, where $q_0 \in I_M$ and $f_M \in F_M$.

The calculation of two NFA equivalences is demonstrated by an adaptation [17] of Hopcroft and the Kraps algorithm for determining the equivalence of two DFAs [8]. A variation of the algorithm will be used later in Chapter 5 for the sub-approximation of a language equivalence.

An empty set R is initialized to $\emptyset$ at the beginning of the algorithm. R will store the already processed pairs and prevent the algorithm from an infinite loop. Thereafter, the set *todo* will be initialized, and the pair $(I_A,\ I_B)$ is inserted. I_A is a set of initial states of the automaton A and I_B is the set of initial states of the second automaton B. The pair

(X, Y) is selected, while *todo* is not empty. The pair (X, Y) is bad if one state of X or Y is accepting whereas the other is not. If the pair is bad, then automata A and B are not equivalent. Otherwise, the successors of X and Y are generated into *todo*. If the set *todo* is empty, the automata A and B are equivalent.

Algorithm 2.14 The naive algorithm for checking NFAs equivalence

```
    Input: two NFAs $A = (Q_A, \Sigma_A, \delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma_B, \delta_B, I_B, F_B)$
    Output: „Yes“ if $L(A) \equiv L(B)$, otherwise „No“
 1: $R \leftarrow \emptyset$, $todo \leftarrow \{(I_A, I_B)\}$
 2: while $todo \neq \emptyset$ do
 3:     Pick $(X, Y) \in todo$ and remove it
 4:     if $(X, Y) \in R$ then
 5:         continue
 6:     end if
 7:     if $(X, Y)$ is bad pair then
 8:         return „No, $L(A) \neq L(B)$“
 9:     end if
10:     for $a \in \Sigma$ do
11:         $todo \leftarrow todo \cup \{(\delta_A(X, a), \delta_B(Y, a))\}$
12:     end for
13:     $R \leftarrow R \cup \{(X, Y)\}$
14: end while
15: return „Yes $L(A) \equiv L(B)$“
```

2.3 States Languages

To minimize the states of NFA, it is necessary to know the relations between their languages. An equivalence relation is the most commonly used relation for automata minimization. It can be an equivalence of forward, backward, or both of these languages. Equivalent states inform about the duplication of machine subgraphs of which they are apart. The language equivalent states can be merged. Another important relation is language inclusion. It can also be an inclusion of *forward languages*, *backward languages*, etc. The language inclusion informs about the similarity of two (or more) states, where one is strong and the other weak. In this case, the weak state will be reduced and the strong will survive. The special languages for the following work will be introduced. These languages are interstate language, pure language, and maximal distance language. *Interstate language* is defined by an oriented route between two states. The *pure language* can be any already defined language, but the routes which define this language cannot lead through a specified (forbidden) state. The last newly introduced language markup will be the *maximal distance language*. This language is defined by a particular state and its ancestor or successor (depends on the type of a language) to a maximal distance.

The basic group of state languages consists of forward and backward languages. Backward language is defined by a set of strings over the automaton alphabet, for which exists a sequence of transitions (route) from an initial state to an examined state. We will write the backward language of state $q \in Q$ as $\overleftarrow{L}(q)$, or simply $\overleftarrow{q}$.

Definition 2.15 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***backward language of the state*** *$q \in Q$ is defined as $\overleftarrow{L}(q) = \{w_b \,|\, w_b \in \Sigma^*,\ q_0 w_b \vdash^* q,\ \text{where } q_0 \in I\}$.*

On the contrary, the forward language of the state is the set of strings, for which exists the route from the actual state to the final state. We will write the forward language of the state $q \in Q$ as $\overrightarrow{L}(q)$, or simply $\overrightarrow{q}$.

Definition 2.16 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***forward language of the state*** *$q \in Q$ is defined as $\overrightarrow{L}(q) = \{w_f \,|\, w_f \in \Sigma^*,\ q w_f \vdash^* f,\ \text{where } f \in F\}$.*

The interstate language is the first language that belongs to the special group of languages defined for this thesis. The language, as already has been said, is defined by the set of strings over the alphabet of the automaton, for which exists a sequence of transitions between the first and second state (*boundary states*). We will write the language between states q and p as $L(q, p)$.

Definition 2.17 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***interstate language*** *between two states p and $q \in Q$ is defined as $L(p, q) = \{w_i \,|\, w_i \in \Sigma^*,\ p w_i \vdash^* q\}$.*

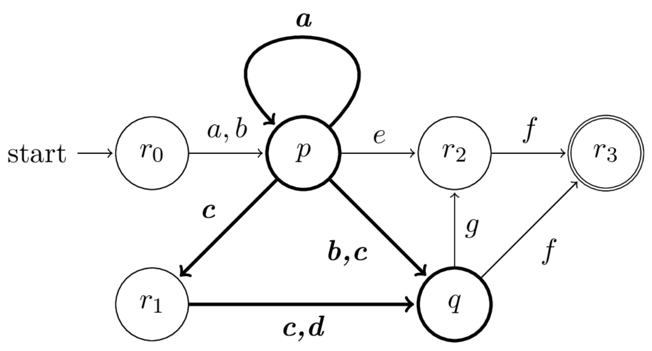


Figure 2.18: Interstate language $L(p, q)$ is defined by the bold transitions.

Another special language is pure language. This language is defined by the set of strings over an automaton alphabet, for which there exists a sequence of transitions between boundary states (initial and actual state for the backward language, examined state and final states for the forward language, or between two states for the interstate language) without the usage of the forbidden state. This condition does not apply to the boundary states themselves. For example, the pure backward language of the state q, which does not use the forbidden state r, is a set of strings, for which exists a route from an initial state to state q without using r.

Definition 2.19 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***pure backward language*** *of the state $q \in Q$, denoted $\overleftarrow{L}(q, \overline{s})$, is defined as the set of all strings w in the form $w = y_1 y_2 \ldots y_n$, where $y_i \in \Sigma$ for $i = 1 \ldots n$, for which exists such a sequence of states $r_0 r_1 \ldots r_n$, where $r_0, r_n \in Q \setminus \{s\}$ and $r_i \in Q$ for $i = 1 \ldots n-1$, with three conditions:*

1. *$r_0 \in I$,*
2. *$r_{i+1} = \delta(r_i, y_{i+1})$ for $i = 0 \ldots n-1$, and*

3. *$r_n = q$.*

Definition 2.20 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then **pure forward language** of the state $q \in Q$, denoted $\overrightarrow{L}(q, \overline{s})$, is defined as the set of all strings w in the form $w = y_1 y_2 \dots y_n$, where $y_i \in \Sigma$ for $i = 1 \dots n$, for which exists such a sequence of states $r_0 r_1 \dots r_n$, where $r_0, r_n \in Q \setminus \{s\}$ and $r_i \in Q$ for $i = 1 \dots n-1$, with three conditions:*

1. *$r_0 = q$,*

2. *$r_{i+1} = \delta(r_i, y_{i+1})$ for $i = 0 \dots n-1$, and*

3. *$r_n \in F$.*

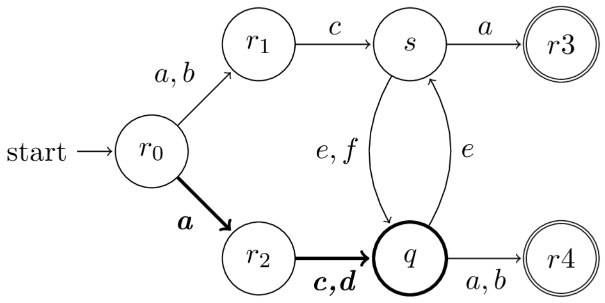


Figure 2.21: Pure backward language $\overleftarrow{L}(q, \overline{s})$ is created by the bold transitions.

Definition 2.22 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then **pure language between states** q and $p \in Q$, denoted $L(p, q, \overline{s})$, is defined as the set of all strings w in the form $w = y_1 y_2 \dots y_n$, where $y_i \in \Sigma$ for $i = 1 \dots n$, for which exists such a sequence of states $r_0 r_1 \dots r_n$, where $r_0, r_n \in Q \setminus \{s\}$ and $r_i \in Q$ for $i = 1 \dots n-1$, with three conditions:*

1. *$r_0 = p$,*

2. *$r_{i+1} = \delta(r_i, y_{i+1})$ for $i = 0 \dots n-1$, and*

3. *$r_n \in q$.*

The last special language is a language with a maximal distance. It is a standard language enriched with the feature that the strings which define the language are defined only by the state with maximal distance n from the current state. A language with a maximal distance will be widely used for the forward and backward language specification. Backward language with maximal distance is defined as suffixes of all strings from the default backward language. Conversely, prefixes of all strings from the old language define the forward language with maximal distance.

Definition 2.23 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $\overleftarrow{L}(q)$ the backward language of the state $q \in Q$. Then **backward language with maximal distance** $n \in \mathbb{N}$ is defined as $\overleftarrow{L_n}(q) = \{w' \mid |w'| \leq n, \ \exists x \in \Sigma^* \text{ for which } xw' \in \overleftarrow{L}(q)\}$*

Definition 2.24 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $\overrightarrow{L}(q)$ the forward language of the state $q \in Q$. Then* ***forward language with maximal distance*** *$n \in \mathbb{N}$ is defined as $\overrightarrow{L_n}(q) = \{w' \mid |w'| \leq n,\ \exists x \in \Sigma^*$ for which $w'x \in \overrightarrow{L}(q)\}$*

All these language behaviors can be combined. We can ask for the pure backward language with a maximal distance. Let show this engaging language by an example.

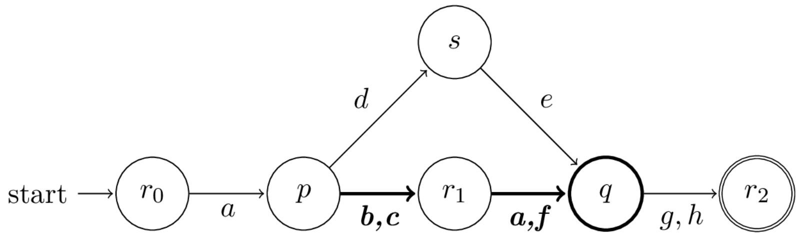


Figure 2.25: Pure backward language of a state q without a state s with maximal distance 2, $\overleftarrow{L_2}(q, \overline{s}) = \{ba, af, ca, cf\}$.

2.4 Simulation

Two methods for calculating a language inclusion, and ultimately equivalence, exist. The first method is based on "subset construction". Subset construction algorithm transforms NFA into DFA, which is then an input of the equivalence checking algorithm. The state explosion might occur during the NFA transformation. If Q is the set of states in the original NFA, the power set $P(Q)$ is of size $2^{|Q|}$, so the DFA may contain up to $2^{|Q|}$ states [27]. Moreover, methods based on the simulation are decidable in polynomial time [14]. They are often more efficient than methods based on subset construction. Therefore, the simulation is rather used in many cases. Unfortunately, the simulation is computationally incomplete and has a stronger relationship with an automaton than the language inclusion. Thus, the simulation implies language inclusion, but not vice versa. Some language inclusions might not be detected by the simulation.

Definition 2.26 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***simulation*** *is a relation $\preceq \subseteq Q \times Q$, such that $p \preceq r$ only if:*

1. *$p \in F \implies r \in F$ and*
2. *for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$*

Algorithm 2.27 Naive simulation algorithm [20]

```
    Input: NFA $M = (Q, \Sigma, \delta, I, F)$
    Output: for each state $q \in Q$, the simulation $sim(q)$
1: for $q \in Q$ do
2:     $sim(q) \leftarrow \{r \mid \delta(r, a) \subseteq \delta(q, a)\ \forall a \in \Sigma\}$
3: end for
4: while there are three state $q$, $r$ and $s$ such that $q \in succ(r)$, $s \in sim(q)$,
5:         and $succ(w) \cap sim(r) = \emptyset$ do
6:     $sim(q) \leftarrow sim(q) \setminus \{s\}$
7: end while
```

Chapter 3

Existing Minimization Techniques

Automata and their parts, on which the work focuses, can also be minimizable by existing techniques, but these techniques leave potentially minimizable subgraphs in the automata. State merge [6, 10, 24, 22], transition pruning [10, 13], and transition adding (saturation) [6, 13] are so far the most effective minimization methods. Their mutual use merges equivalent states and creates useless states, which allows their future elimination.

Despite their minimizing power, they are not omnipotent. There are types of automata, or their parts, that can be easily minimized, sometimes on the first look. However, all existing approaches based on language equivalence or inclusion relations are helpless. Example of a nonminimizable automaton is in the last Section of this chapter.

3.1 States Merging

The most well-known minimization method of nondeterministic finite automata is state merging [6, 10, 24, 22]. The technique merges two states based on their language equivalence. It might be forward, backward, or both sides equivalence. The merging can be done even on the basis of bilateral language inclusion. Conditions for states merge are declared by Theorem 3.2 [22]. The equivalence can be approximated by the simulation relation. If two states p and q are equivalent, then they will be merged into a new state m. Within the merge, all transitions which lead through the source states p and p are redirected to the destination state m.

Definition 3.1 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $p, q, m \in Q$. The **automaton with merged states** p and q into m is $M' = (Q', \Sigma, \delta', I', F')$, where*

1. $Q' = (Q \setminus \{p, q\}) \cup \{m\}$ *is a finite set of states,*

2. Σ *is an alphabet,*

3. $$\delta'(s, a) = \begin{cases} \delta(p, a) \cup \delta(q, a) & \textit{for } s = m, \\ (\delta(s, a) \setminus \{q\}) \cup \{m\} & \textit{for } q \in \delta(s, a), \\ (\delta(s, a) \setminus \{p\}) \cup \{m\} & \textit{for } p \in \delta(s, a), \\ \delta(s, a) & \textit{otherwise.} \end{cases}$$

4. $$I' = \begin{cases} (I \setminus \{p, q\}) \cup \{m\} & \textit{if } p \vee q \in I, \\ I & \textit{otherwise.} \end{cases}$$

5. $F' = \begin{cases} (F \setminus \{p, q\}) \cup \{m\} & \textit{if } p \vee q \in F, \\ F & \textit{otherwise.} \end{cases}$

Theorem 3.2 [22] *Two states p and q from the automaton M can be merged if at least one of the following conditions is met:*

1. $\overleftarrow{L}(p) \subseteq \overleftarrow{L}(q) \wedge \overleftarrow{L}(p) \supseteq \overleftarrow{L}(q)$,
2. $\overrightarrow{L}(p) \subseteq \overrightarrow{L}(q) \wedge \overrightarrow{L}(p) \supseteq \overrightarrow{L}(q)$, *or*
3. $\overleftarrow{L}(p) \subseteq \overleftarrow{L}(q) \wedge \overrightarrow{L}(p) \subseteq \overrightarrow{L}(q)$.

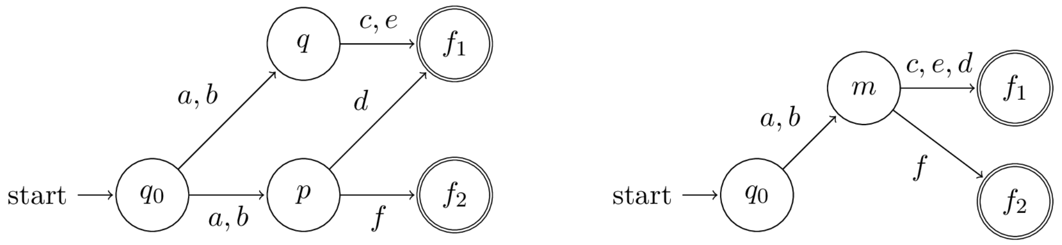


Figure 3.3: Automaton $M_{3.3}$ (on the left) and its minimized version with states p and q merged into m.

3.2 Transition Pruning

The basic idea of transition pruning [10, 13] is the existence of a better transition (stronger language), which can overtake the function of the deleting transition. Definitions and theorems are simplification of [13].

Definition 3.4 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. Then* ***automaton with pruned transition*** *$p \xrightarrow{b} r$, where $p, r \in Q$ and $b \in \Sigma$, is $M' = (Q, \Sigma, \delta', I, F)$ write as $Prune(M, p \xrightarrow{b} r)$ where:*

1. *Q is a finite set of states,*
2. *Σ is an alphabet,*
3. $\delta'(s, a) = \begin{cases} \delta(s, a) \setminus \{r\} & \textit{for } s = p \wedge a = b, \\ \delta(s, a) & \textit{otherwise.} \end{cases}$
4. *$I \subseteq Q$ is a finite set of initial states, and*
5. *$F \subseteq Q$ is a finite set of final states.*

The following theorems show the use cases where the transition pruning can be used. The proofs of the theorems are shown on Büchi word automata (NBA), in [13, p.16–20].

Theorem 3.5 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, $p, q, r \in Q$ and $a \in \Sigma$. The transition $r \xrightarrow{a} p$ can be pruned if there exist $r \xrightarrow{a} q$ and $\overrightarrow{L}(p) \subseteq \overrightarrow{L}(q)$.*

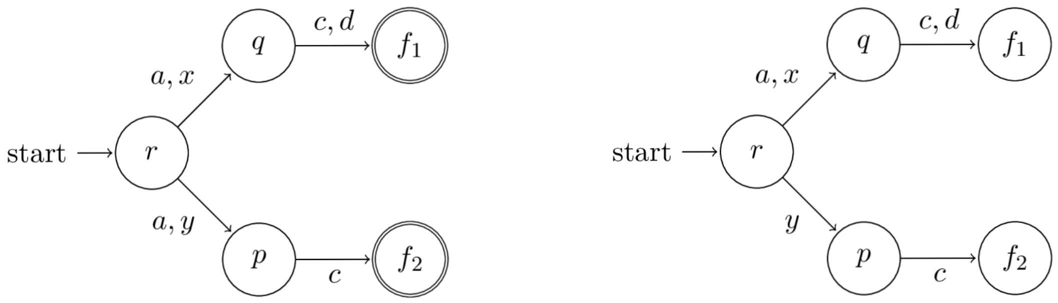


Figure 3.6: Automaton $M_{3.6}$ (on the left) and automaton $Prune(M_{3.6}, r \xrightarrow{a} p)$ (on the right). Transition was pruned according to *Theorem 3.5*.

Theorem 3.7 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, $p, q, r \in Q$ and $a \in \Sigma$. The transition $p \xrightarrow{a} r$ can be pruned if there exist $q \xrightarrow{a} r$ and $\overleftarrow{L}(p) \subseteq \overleftarrow{L}(q)$.*

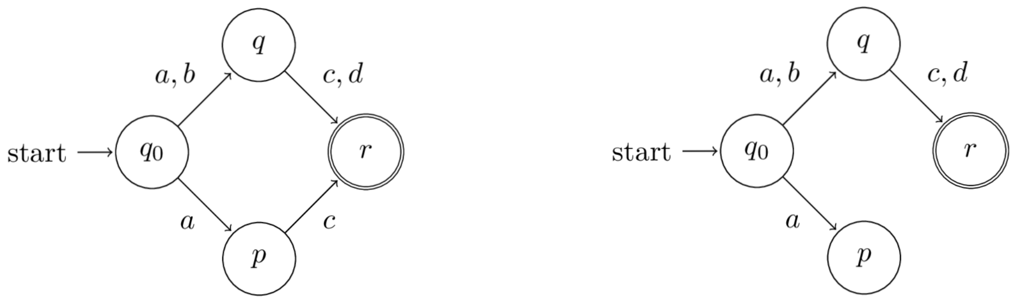


Figure 3.8: Automaton $M_{3.8}$ (on the left) and automaton $Prune(M_{3.8}, p \xrightarrow{a} r)$ (on the right). Transition was pruned according to the *Theorem 3.7*.

Theorem 3.9 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, $p, q, p', q' \in Q$ and $a \in \Sigma$. The transition $q' \xrightarrow{a} p'$ can be pruned if there exist $p \xrightarrow{a} q$ and $\overleftarrow{L}(p') \subseteq \overleftarrow{L}(p) \wedge \overrightarrow{L}(q') \subseteq \overrightarrow{L}(q)$.*

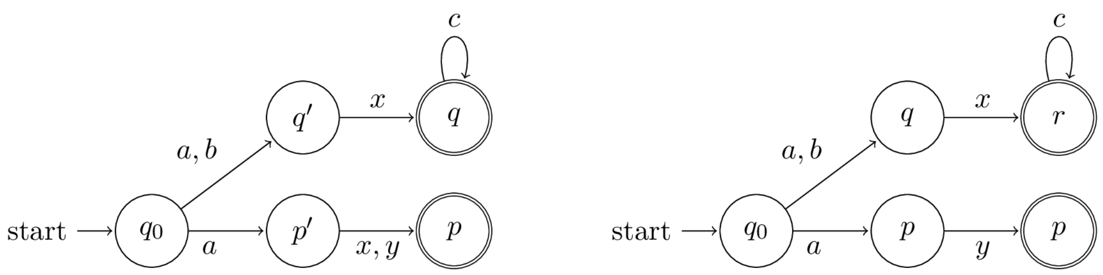


Figure 3.10: Automaton $M_{3.10}$ (on the left) and automaton $Prune(M_{3.10}, q \xrightarrow{a} p')$ (on the right). Transition pruning was based on the *Theorem 3.9*.

3.3 Saturation

The saturation [6, 13] allows the addition of the new transition without changing the automaton language. Saturation is used as an extension of merge and transition pruning that are applied during the standard minimization process. At some point, it is no longer possible to minimize the automaton only by using merge and transition pruning. Thanks to saturation and addition of new transitions, the future minimization is possible. Saturation is made only if a particular transition already exists. It is an analogy of transition pruning. Definitions and theorems are simplification of [13].

Forward saturation is done only when there exist two states in backward language inclusion. The saturated state is enriched with all forward transitions of the stronger state to its successors. An example of forward saturation is in Figure 3.15.

Definition 3.11 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $p, q \in Q$. Then the **forward saturation** of the state p by state q, write $\overrightarrow{Sat}(M, q, p)$, change the transition function δ as follows:*

$$\delta'(s, a) = \begin{cases} \delta(p, a) \cup \delta(q, a) & \textit{for } s = p, \\ \delta(s, a) & \textit{otherwise.} \end{cases}$$

Theorem 3.12 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $p, q \in Q$. The forward saturation $\overrightarrow{Sat}(M, q, p)$ can be done if $\overleftarrow{L}(p) \subseteq \overleftarrow{L}(q)$.*

Backward saturation is done only when there exist two states in forward language inclusion. The saturated state is enriched with all transitions incoming to a stronger state from its ancestors.

Definition 3.13 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $p, q \in Q$. Then the **backward saturation** of the state p by state q, write $\overleftarrow{Sat}(M, q, p)$, change the transition function δ as follows:*

$$\delta'(s, a) = \begin{cases} \delta(s, a) \cup \{q\} & \textit{for } p \in \delta(s, a), \\ \delta(s, a) & \textit{otherwise.} \end{cases}$$

Theorem 3.14 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $p, q \in Q$. The forward saturation $\overleftarrow{Sat}(M, q, p)$ can be done if $\overrightarrow{L}(p) \subseteq \overrightarrow{L}(q)$.*

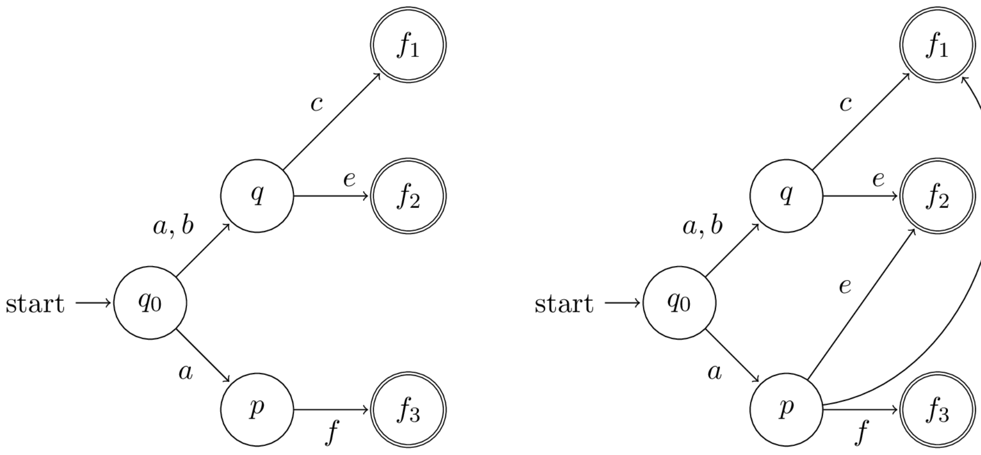


Figure 3.15: Automaton $M_{3.15}$ (on the left) and automaton $\overrightarrow{Sat}(M_{3.15}, q, p)$ (on the right).

3.4 Limitation of Existing Methods

Although the current methods are very robust, they are not almighty. There still exist subgraphs of the automaton that they cannot solve (minimize). In some cases, it is a simply minimization visible to the naked eye. However, the methods, such as state merging, transition pruning, and saturation fail. The following paragraph demonstrates an example of an automaton that cannot be minimized by current methods.

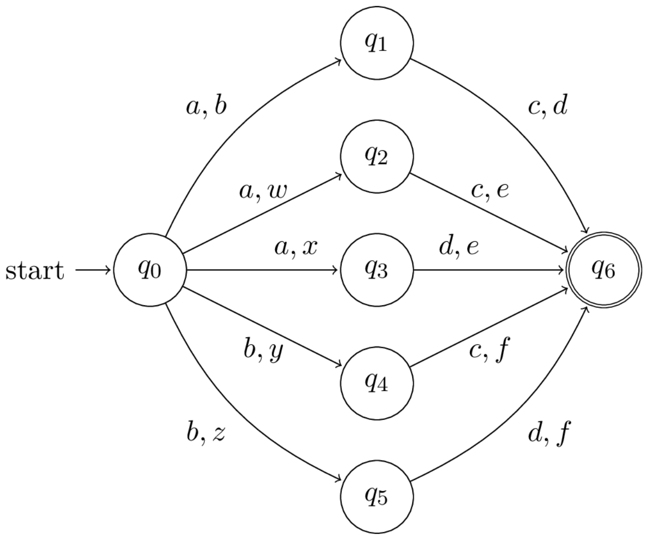


Figure 3.16: Automaton $M_{3.16}$ is not minimizable by existing minimization methods (merge, saturation, neither transition pruning).

The oldest minimization method, state merging, cannot be applied on the automaton in Figure 3.16, because the merging requires the existence of two equivalent states that could be merged. However, there is not any equivalent state. At the same time, it is not possible to saturate the automaton, because the saturation requires a relation of language inclusion between a pair of states. There is no such pair. Transition pruning cannot be applied, because it requires the existence of language inclusion too. We can see that any of the current methods cannot be used for minimization of the automaton from Figure 3.16.

It is certainly obvious to the attentive reader, that the language over the state q_1 is entirely covered by the language of the set of states $\{q_2, q_3, q_4, q_5\}$. Therefore, it is unnecessary for the language of the automaton. The state q_1 can be eliminated. The minimization was very trivial, but known methods were not able to do it.

Chapter 4

Initial Observation

This work aims to develop a method for reducing the size of an automaton based on the coverage of a part of the language of a state by a set of other states. The method will focus only on local parts of the automaton with common successors or common ancestors. The minimized fragments cannot have any language inclusion, therefore the existing minimization methods, such as state merging, transition pruning, and saturation cannot reduce them. The ideas which can provide minimization of so far nonminimizable parts of the automata, or based on which the standard minimization results can be optimized are presented in this chapter.

4.1 Focused Fragments

The investigated minimization approach will focus only on sets of states, where a part of a language of some states is represented yet by the others.

The basic case of such a set is a subgraph with a single ancestor (source) and single successor (target). This is a subgraph 1:1. Another case is a subgraph with relation 1:N, respectively, N:1. The automaton $M_{4.1}$ shows a subgraph 1:N with only one common source and more than one successor (target) of the set S. With an N:1 relationship, the situation is the opposite. There is also a general relation N:N with many ancestors and many successors. All cases are subjects of the research of the new minimization method in this thesis.

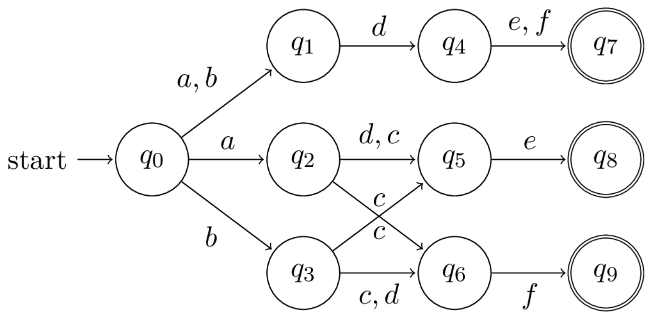


Figure 4.1: Automaton $M_{4.1}$ with relation 1:N over the set of states $\{q_1, q_2, q_3\}$.

4.2 Naive Approach

Each *focused fragment* can be easily reduced according to the number of letters of incoming and outcoming transitions. Example of this minimization of the automaton from Figure 3.16 is shown below. Because the original outcoming transitions of the set $S = \{q_1, q_2, q_3, q_4, q_5\}$ leads to the common successor with 4 different characters, the number of transitions (and states of a set S) can be decreased to this number. After that, the transition edges going from the common ancestor to the states of the set S must be recalculated to keep the language of the automaton unchanged. The number of states is reduced by one and transitions are reduced by one too. However, from the previous Section, we know that there is a more efficient result with only 16 transitions.

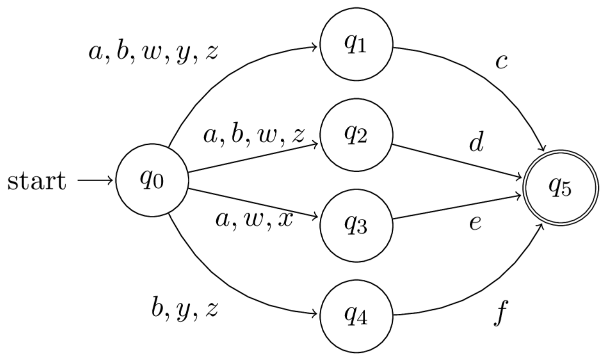


Figure 4.2: Naive minimization of the automaton from Figure 3.16.

4.3 Remerging Approach

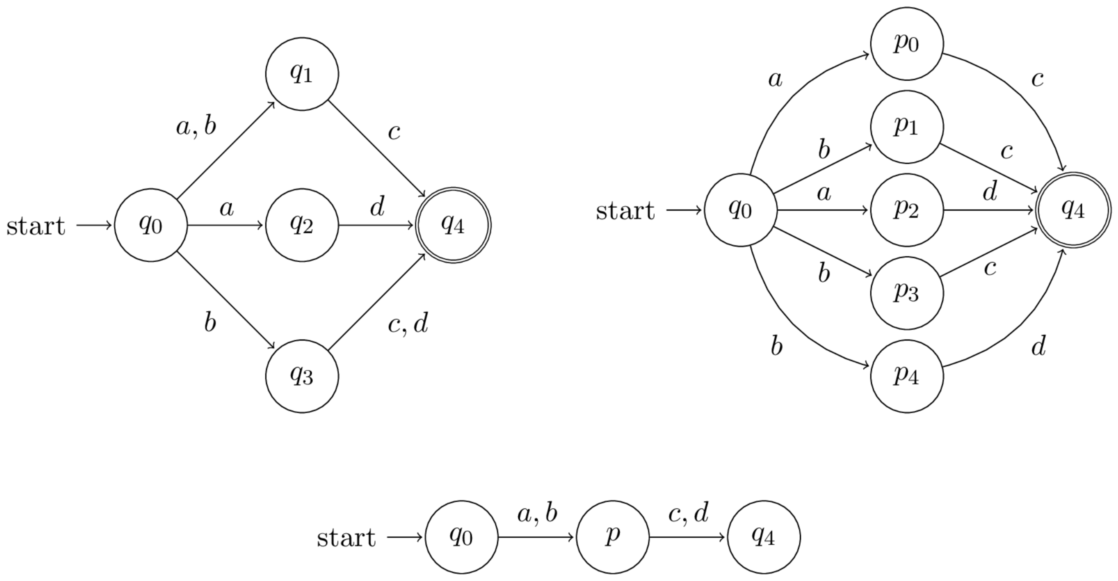


Figure 4.3: Automaton $M_{4.3}$ (on the top left), its simplified version (on the top right), and reduced result (on the bottom).

The example of an automaton where merge nor transition pruning cannot be used is shown in Figure 4.3. Saturation can be used and can potentially add some transitions, that can make merge or transition pruning usable. But let us show the reduction of such an automaton without the saturation. The input automaton does not have any language equivalent state nor states in bilateral language inclusion. The merge and transition pruning cannot be used. If we increase the size of the automaton by adding states that will overtake some parts of languages of the strongest states (q_1 and q_3), it can be seen that many language equivalences occur now. It is necessary to merge states in a correct order. Merging of states p_0 and p_2 based on the backward language equivalence, and then the states p_1, p_3, and p_4 based on the backward language equivalence allows a future merge of state $p_{0,2}$ (created by merge of p_0 and p_2) and state $p_{1,3,4}$. This merge order gives the most optimal result. On the contrary, if we merge states p_1 and p_3 based on the backward language equivalence and then states p_2 and p_4 based on the forward language equivalence, we get stuck. No more merges can be performed on such the automaton. We get a sub-optimal solution with one more state. Let us call the problem of the maximization of performed merge a *the problem of optimal merge* defined in 5.22.

4.4 The Problem of Optimal Merge

The optimal order of state merging, based on an equivalence, does not exist [23]. Various orders can have different results, with the same language.

During the merge, the state can be in a backward language equivalence relation with a set of states and a forward language equivalence relation with another set of states. Lemmas say that it is safe to merge states only by one language equivalence. The newly created stat will have the same language (backward for merge performed based on the backward language equivalence and forward for merge based on the forward language equivalence). And new merging does not need recalculation of the language relations.

Lemma 4.4 *If $\overleftarrow{L}(p) \equiv \overleftarrow{L}(q)$ and states p and q are merged into m, then $\overleftarrow{L}(m) \equiv \overleftarrow{L}(p) \equiv \overleftarrow{L}(q)$.*

If the states p and q were merged into the state m base on the backward language equivalence, then the newly created state m can be used in the next backward merging based on the old calculations of language equivalence instead of states p or q, but not in the merging based on the forward language equivalence. The language can be changed.

Lemma 4.5 *If $\overrightarrow{L}(p) \equiv \overrightarrow{L}(q)$ and states p and q are merged into m, then $\overrightarrow{L}(m) \equiv \overrightarrow{L}(p) \equiv \overrightarrow{L}(q)$.*

If the states p and q were merged into the state m base on the forward language equivalence, then the newly created state m can be used in the next forward merging based on the old calculations of language equivalence instead of states p or q, but not in the merging based on the backward language equivalence. The language can be changed.

The merge of states p and q based on the bilateral language inclusion alway block newly created state m from further merge (backward and forward), based on the old calculation of language inclusions, because the language (backward and forward) of a state m might not be equivalent to the language of the state p nor q as shown below. For this reason, the investigated reduction approach does not use the bilateral language inclusion for merge.

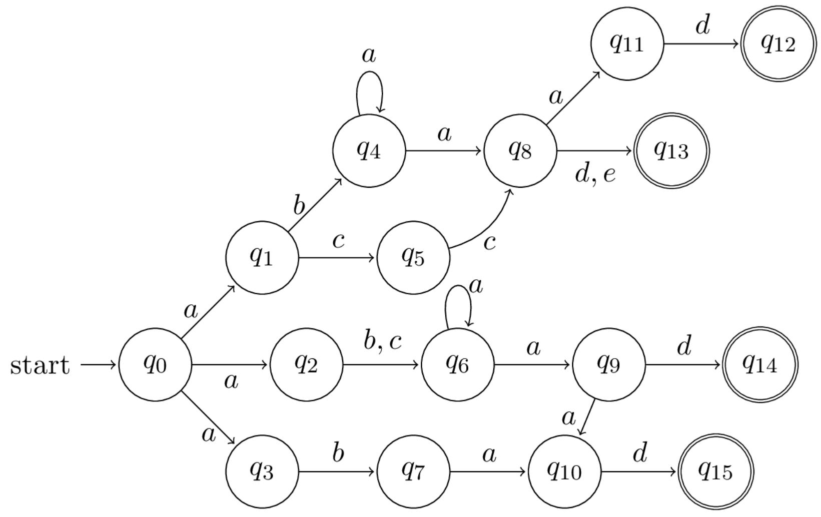


Figure 4.6: The NFA $M_{4.6}$, the states q_9 and q_{10} will be merged.

Automaton $M_{4.6}$ shows that it is possible to arbitrarily merge either states q_8 and q_9 or states q_9 and q_{10}. However, what may not be obvious at first glance is the impossibility of merging states q_9 and q_{10} first and then states q_8 and q_9.

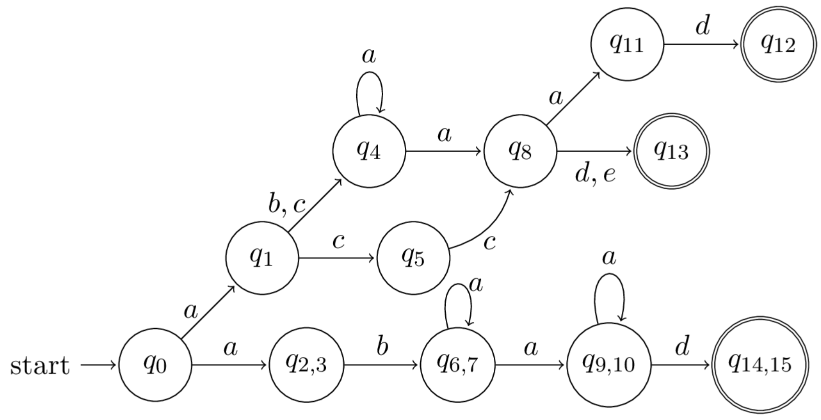


Figure 4.7: Automaton $M'_{4.6}$, after small adjustments and merging of q_{10} and q_9 in Automaton $M_{4.6}$.

Automaton $M'_{4.6}$ demonstrates the inequivalence of a language of states q_9 or q_{10} and the merge result $q_{9,10}$. The new state $q_{9,10}$ can not be used in the next merge without recalculation of a language relations. If we try to merge q_8 and q_9 (q_9 is $q_{9,10}$ after merge) on Automaton $M'_{3.10}$, the language will be changed.

Because some states can be in forward and backward language equivalence with other states, it is hard to predict the most optimal order of state merging to get a maximal merged states. The main question is: “According to which equivalence merge states that are in backward and forward language equivalence, to get the most optimal result?„ The technology that will answer this question is SAT solver, which will be used to determine the most appropriate merging procedure. The solver information will be the cornerstone for *SAT-solver-based automata minimization* algorithm, which is described in the following chapter.

Chapter 5

SAT Solver in Minimization

This Chapter presents the main methods used in the minimization of nondeterministic finite automata using SAT solver. The SAT-solver-based approach minimizes the automaton by parts (subgraphs), which consists of a group of states with a common ancestors or common successors. The set of these states is called a family. The definition and family lookup algorithm are described in Section 5.1. To get the automaton from its local minimum and allow a more minimal solution, it is necessary to replace this family with a new set of states which has the same language, but each state has maximal one incoming and one outcoming transition. This multiplication process is presented in Section 5.2. After the multiplication, the language relations of a new multiplied state of a family are coded for SAT solver. The special approximation of a language equivalence is used in SAT-solver-based automata minimization. The approximation algorithm is defined in Section 5.3. The solver gives information on how to merge states of the family so that the final number of merged states is maximal. The coding principles and an example are shown in Section 5.4. The minimization algorithm using approaches such as family selection, state multiplying, and solver task coding, is defined in Section 5.5.

5.1 Family of States

The algorithm using the SAT solver reduces the automaton by parts which consists of a set of states (*family*) with a common successors or common ancestors and with a same transition letter. Only two basic groups of states (*proto-families*) and then their combinations will be minimized.

The first group is defined by a common ancestor which marks as a proto-family the states into which leads similarly notated transitions. These transitions create nondeterminism. That means that there is a state in which a backward language (or a piece of a backward language) is already covered by another state of this proto-family. This condition does not apply to the ancestor itself. Common ancestor of the family cannot be its member. States of the family cannot have any direct transitions between each other. The state of the proto-family from or to which leads a transition from another state of this proto-family must be eliminated from the proto-family. The proto-family with a common ancestor is called a *forward proto-family*, or simply 1:N.

Definition 5.1 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, state $s \in Q$, and letter a in Σ. The* ***forward proto-family*** *(1:N) of the state s and the letter a is a set $P \subseteq Q$ such that:*

- $q \in P \implies s \xrightarrow{a} q \in \delta$,
- $p, q \in P \implies \nexists b \in \Sigma$ *such that* $p \xrightarrow{b} q \in \delta$,
- *P is a maximal set satisfying 1 and 2.*

The complement of the 1:N proto-family is a proto-family with a common successor. All states from which lead a transition into the common successor with the same assignment belong to the same proto-family. The backward nondeterminism indicates that the forward language of some state of the proto-family might be already represented by another state of this proto-family. As in the previous definition, this condition does not apply to the successor itself. Common successors of the proto-family cannot be its member. States of the proto-family cannot have any direct transition between each other. This proto-family is called a backward proto-family, or simply N:1.

Definition 5.2 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA, state $s \in Q$, and letter a in Σ. The* ***backward proto-family*** *(N:1) of the state s and the letter a is a set $P \subseteq Q$ such that:*

- $q \in P \implies q \xrightarrow{a} s \in \delta$,
- $p, q \in P \implies \nexists b \in \Sigma$ *such that* $p \xrightarrow{b} q \in \delta$,
- *P is a maximal set satisfying 1 and 2.*

Definition 5.3 *The* ***pre-family*** *is 1) family, 2) union of proto-families which have intersection, 3) nothing else.*

Definition 5.4 *The* ***family*** *is a maximal subset of pre-family, such that none of the family states has a transition between them.*

In the special cases where the family has only one ancestor or successor, the number of states needed for a replacement of an existing family can be predicted as a minimum of theorems 5.5 or 5.7.

Theorem 5.5 *The family $F \subseteq Q$ 1:N, with only one ancestor, can be minimized to n states by a redistribution of transitions. Where $n = |\overleftarrow{L}_1(F, \overline{F})|$ is the cardinality of a pure backward language with maximal distance 1 of the family F withou using the states of F.*

Based on the theorem 5.5, if the family $F \subseteq Q$, consists of 4 states, has only one successor $s \in Q$ and the transitions going from the successor to states of the family are assigned by the letters $a, b, c \in \Sigma$ (the size of pure backward language with maximal distance 1 of the family is 3), then the family could be minimized to 3 states. Each new state will handle one incoming letter. Outcoming transitions of states of the family must be recombined to cover the original language. This example of recombination is shown in Figure 5.6. If the family satisfies both theorems (the family has only one ancestor and only one successor), then the family can be minimized to the minimum count of states from both theorems.

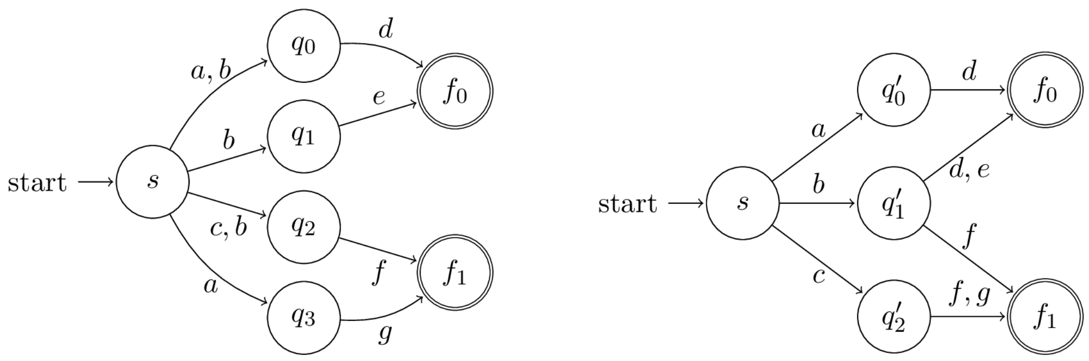


Figure 5.6: Automaton on the left has a family consisting of states $F = \{q_0, q_1, q_2, q_3\}$. $|\overleftarrow{L}_1(F, \overline{F})| = 3$. That means that the family can be minimized to 3 states.

Theorem 5.7 *The family $F \subseteq Q$ N:1, with only one successor, can be minimized to n states by a redistribution of transitions. Where $n = |\overrightarrow{L}_1(F, \overline{F})|$ is the cardinality of the pure forward language with maximal distance 1 of the family F without using the states of F.*

Some states can be assigned to more than one family. Take an example from Figure 5.6. The states q_0 and q_3 are in the forward family, created by an ancestor s and a transition letter a. The second forward family consists of states q_0, q_1, and q_2, created by an ancestor s and a transition letter b. It is necessary to join the families with a common state, to get the correct and the most useful information about languages of states in a family. Here, the state q_0 is in both families. Therefore, these two families will be joined into one. Sometimes, the state can be in more forward (1:N) and backward (N:1) families.

The problem of joining sets (families) with a common element (state) can be easily solved in graph theory as a problem of finding connected components. Algorithm[1] 5.8 describes an approach for joining sets with a common element. This algorithm will be used not only for merging a set of families with a common element, but for merging a set of equivalent pairs of states with a common state.

The algorithm creates equivalence classes of transitive and reflexive closure of the neighbour relation on states, where states are neighbors if they appear together in some of the input sets. The algorithm creates interconnections between elements (vertices) of each set (proto-family). After the initialization of a graph, the walk from each vertex is made. Walks detect isolated graph components. The isolated graph component stands for a joined set (family). The result is a set of graph components (set of joined families).

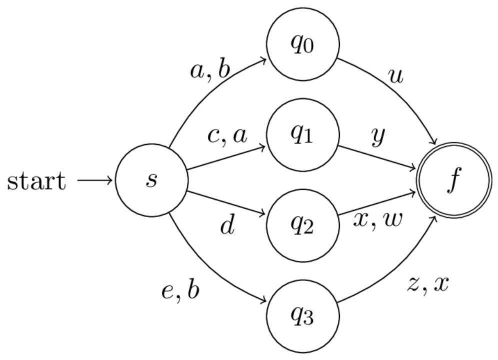
[1] algorithm is taken from `https://www.geeksforgeeks.org/python-merge-list-with-common-elements-in-a-list-of-lists/`

Algorithm 5.8 The algorithm for joining sets with a common element

Input: a S of sets to join.

Output: a set with joined sets.

```
1: $joinedSets \leftarrow \emptyset$
2: $G \leftarrow (V, E)$                                        ▷ unoriented graph
3: $V = \bigcup S$
4: $E = \{\{v_1, v_2\} \mid \forall T \in S : (v_1, v_2) \in T \times T\}$
5: for $vertex \in V$ do                                    ▷ make walks in the graph
6:     $graphComponent \leftarrow \emptyset$
7:     if $vertex \notin visited$ then
8:         $openV \leftarrow \{vertex\}$
9:         while $openV \neq \emptyset$ do
10:            $tmpVertex \leftarrow openV.pop()$
11:            $visited \leftarrow visited \cup \{tmpVertex\}$
12:            $openV \leftarrow openV \cup \{v_2 \mid \{vertex, v_2\} \in H\} \setminus visited$
13:            $graphComponent \leftarrow graphComponent \cup \{tmpVertex\}$
14:        end while
15:        $joinedSets.add(graphComponent)$
16:    end if
17: end for
18: return $joinedSets$
```



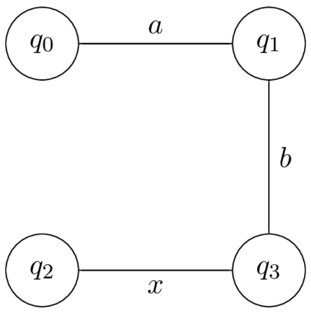


Figure 5.9: Two forward proto-families by a letter a (q_0, q_1) and by a letter b (q_0, q_3) and one backward proto-family by a letter x (q_2, q_3) are in the given automaton. The graph on the right represents a neighbor relation between states (the edge notation is additional). The graph contains only one isolated component (family). All families will be joined in one (q_0, q_1, q_2, q_3).

After joining the classical proto-families 1:N or N:1, new generalized types of families can occur. The most general type of the family is a family with more common ancestors and successors signed as N:N. A special type of a family is a family with only one ancestor and one successor, this is the family 1:1. All these types are shown in Figure 5.10.

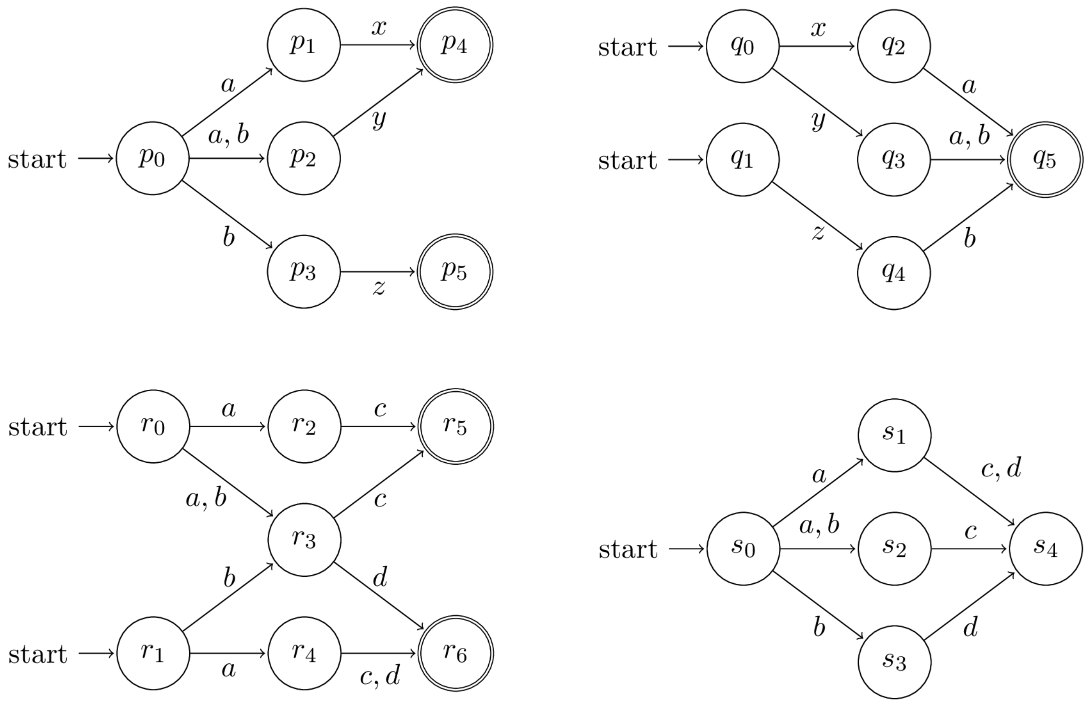


Figure 5.10: (From left to right and top to bottom). The top left automaton M_p represents a forward family 1:N (states p_1, p_2, and p_3). The top right automaton M_q displays an opposite family, the backward family N:1 (of states q_2, q_3, and q_4). The bottom left automaton M_r, shows a combination of forward and backward families, the family N:N (of state r_2, r_3, r_4). The simplest case of a combination of a forward and backward families is the family 1:1 viewed by an automaton M_s in the bottom right.

The family finding algorithm creates for each state the forward and backward proto-family according to the definitions. The forward proto-family of state s is created for each letter from forward transitions. The state s itself is not included in proto-family. The state q is removed from the proto-family if from this state leads a transition to the other state of the proto-family. The backward proto-families of a state s are created similarly. All families with only one state are discarded. After that. Families with a common state are joined by Algorithm 5.11. At the end, the set of families contains the biggest 1:N, N:1, 1:1, and N:N families.

Algorithm 5.11 The algorithm for finding families of states

```
    Input: NFA $M = (Q, \Sigma, \delta, I, F)$.
    Output: a set of families.
1: $families \leftarrow \emptyset$
2: for $s \in Q$ do
3:     for $a \in \Sigma$ do
4:         $successors \leftarrow \emptyset$
5:         $ancestors \leftarrow \emptyset$
6:         for $succ \in \delta(s, a) \setminus \{s\}$ do    ▷ for all successors of a state $s$ by a letter $a$
7:             if $q \notin \delta(succ, y) \cup \delta^{-1}(succ, y)$ where $y \in \Sigma$ and $succ \in successors \setminus \{q\}$ then
8:                 $successors \leftarrow successors \cup \{succ\}$
9:             end if
10:        end for
11:        for $anc \in \delta^{-1}(s, a) \setminus \{s\}$ do    ▷ for all ancestors of a state $s$ by a letter $a$
12:            if $q \notin \delta(anc, y) \cup \delta^{-1}(anc, y)$ where $y \in \Sigma$ and $anc \in ancestors \setminus \{q\}$ then
13:                $ancestors \leftarrow ancestors \cup \{anc\}$
14:            end if
15:        end for
16:        if $|successors| > 1$ then
17:            $families.add(successors)$    ▷ add $successors$ as a family to a set of families
18:        end if
19:        if $|ancestors| > 1$ then
20:            $families.add(ancestors)$    ▷ add $ancestors$ as a family to a set of families
21:        end if
22:    end for
23: end for
24: $families \leftarrow joinSetsWithCommonElem(families)$
25: return $families$
```

To decrease the solver running time, it is suitable to split a family into smaller groups according to the language equivalence pairs. States of each group will have a language equivalence relations (backward or forward) only with other states of the same group. This will minimize the problem of optimal merge. The pair of equivalent states will be assigned to the group by sets joining Algorithm 5.8, which takes a union of forward and backward equivalent pairs. For example, a family $\{q_0, q_1, q_2, q_3, q_4, q_5\}$, where the backward language equivalent state pairs are $\{\{q_0, q_1\}, \{q_3, q_4\}\}$ and a forward language equivalent pairs are $\{\{q_0, q_2\}, \{q_3, q_5\}\}$ will be splitted to two subfamilies $\{q_0, q_1, q_2\}$ and $\{q_3, q_4, q_5\}$. Smaller (sub)families make coding easier and solver decisions faster.

5.2 States Multiplication

The SAT solver minimization approach gives information on how to optimally merge states. The states could be merged only if they have equivalent languages. To increase potentially mergeable states (states with equivalent languages), it is necessary to multiply these states.

The multiplication replaces one state with a new set of states. The new states cover the same language as the original state, but have maximal one incoming and one outcoming transition, self-loops do not count. Let the state s has n incoming and n outcoming transitions. Then the state s will be multiplied to n^2 new states. If two states (s_0 and s_1) in a row (s_1 is the successor of s_0) are multiplied, then the number of new states will be n^3. In general, if m is a number of multiplied states in a row, where each has n incoming and n outcoming transitions, then the number of new states will be m^{n+1}. It is very hard, even impossible, to multiply all states in an automaton to satisfy the condition of maximal one incoming and maximal one outcoming transition.

It is suitable to focus on small automaton parts (best with a quadratic complexity), because multiplying all states of an automaton is expensive. The main idea is that the equivalent (or similar) language of states or even an automaton (subgraph) is caused by a connon successor or common ancestor and transitions with the same letter which leads to or from this common state. And that is the main existential condition of the family. Therefore, only families of states are selected for multiplying. The family on which applies the condition of maximal one incoming and maximal one outcoming transition is called the *multiplied family.*

Definition 5.12 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA and $S \subseteq Q$ a family. The family S is a* ***multiplied family*** *if the following condition applies.*

$$\forall s \in S \left(\sum_{a \in \Sigma} |\delta(s, a) \setminus \{s\}| \leq 1 \wedge \sum_{a \in \Sigma} |\delta^{-1}(s, a) \setminus \{s\}| \leq 1 \right) \tag{5.13}$$

In some cases, the multiplication cannot be done so straightforwardly as only a combination of incoming and outcoming transitions. The first case is a self-loop. Each self-loop represents an infinite set of words, so it cannot be transferred to a limited count of states (without a loop), without changing an automaton language. Therefore, self-loops cannot be multiplied. If the state s is multiplied and has a self-loop, then the multiplication is done only for incoming and outcoming transitions without the self-loop. The entire self-loop is duplicated to each newly created state. The next case is the final or initial state. The solution is much easier here. If the original state is final or initial, then each new state created by a multiplication of the original state will be final or initial too.

Let us show why the states with a transition to another state from a family are discarded by a family finding algorithm. If two states r and s of a family are interconnected (no meter the direction), then it is not possible to satisfy the condition of maximal one incoming and maximal one outcoming transition. The multiplying of the states r and s will only increase the count of interconnected pairs.

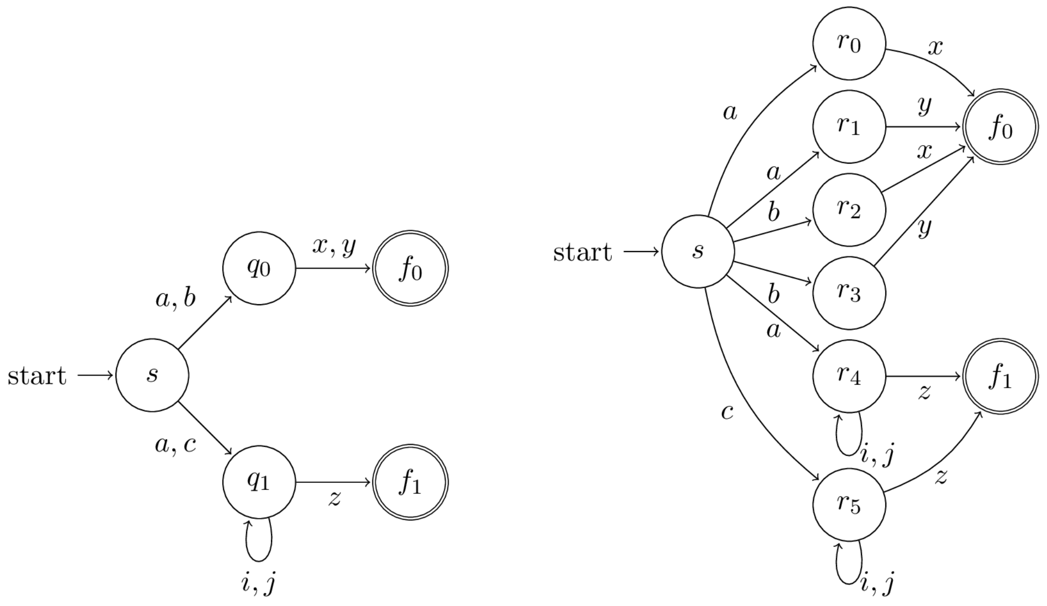


Figure 5.14: The family of states q_0 and q_1 was multiplied. The automaton on the right is the left autaton after family multiplication.

It is not necessary to detect if the states of the family belong to an automaton cycle (*long loop*). Long loops do not have any effect on state multiplication as shown in Figure 5.15. The state ancestor and successor in a long loop behave only as standard ancestors or successors.

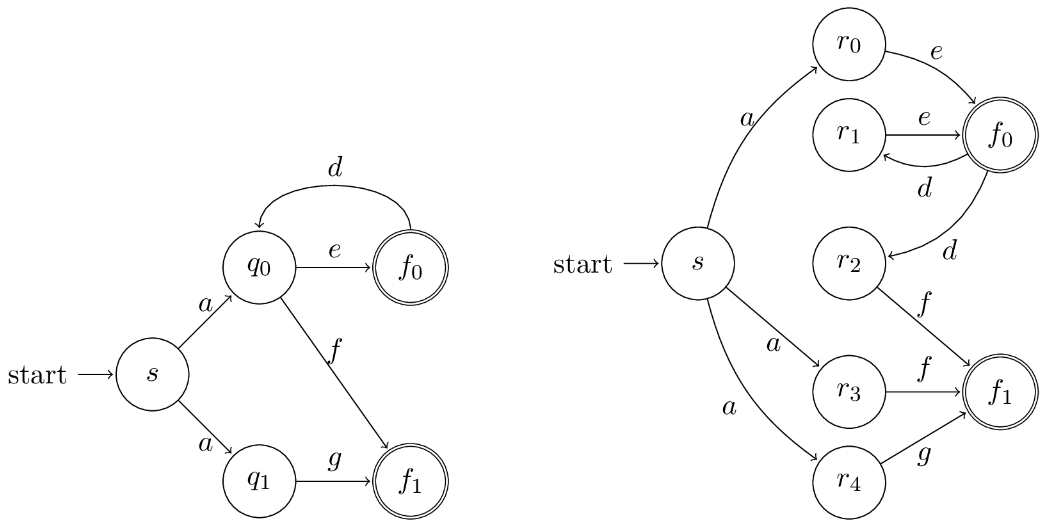


Figure 5.15: The family of states q_0 and q_1 of the left automaton was multiplied. The state q_0 is in the long loop. The automaton on the right shows the situation after multiplication.

Algorithm 5.16 Family of states multiplying algorithm

```
    Input: NFA $M = (Q, \Sigma, \delta, I, F)$, and $family$ (set) of states.
    Output: New family of states created by a multiplication.
 1: $newFamily \leftarrow \emptyset$
 2: for $state \in family$ do
 3:     $possibleAnc \leftarrow anc(state)$
 4:     if $possibleAnc = \emptyset$ then
 5:         $possibleAnc \leftarrow \{\emptyset\}$
 6:     end if
 7:     for $ancestor \in possibleAnc$ do
 8:         $possibleSucc \leftarrow succ(state)$
 9:         if $possibleSucc = \emptyset$ then
10:             $possibleSucc \leftarrow \{\emptyset\}$
11:         end if
12:         for $successor \in possibleSucc$ do
13:             for $bwLetter \in L_1(ancestor, state, \overline{ancestor})$ do
14:                 for $fwLetter \in L_1(state, successor, \overline{state})$ do
15:                     $newState \leftarrow$ a fresh state $\notin Q$
16:                     $newFamily \leftarrow newFamily \cup \{newState\}$
17:                     if $state \in F$ then
18:                         $F \leftarrow F \cup \{newState\}$
19:                     end if
20:                     if $state \in I$ then
21:                         $I \leftarrow I \cup \{newState\}$
22:                     end if
23:                     if $ancestor \neq \emptyset$ then
24:                         $\delta(ancestor, bwLetter) \leftarrow \delta(ancestor, bwLetter) \cup \{newState\}$
25:                     end if
26:                     if $successotr \neq \emptyset$ then
27:                         $\delta(newState, fwLetter) \leftarrow \delta(newState, fwLetter) \cup \{successor\}$
28:                     end if
29:                     for $loopLetter \in L_1(state, state, \overline{state})$ do
30:                         $\delta(newState, loopLetter) \leftarrow \delta(newState, loopLetter) \cup \{newState\}$
31:                     end for
32:                 end for
33:             end for
34:         end for
35:     end for
36:     remove $state$ and its transitions form $M$
37: end for
38: return $newFamily$
```

Algorithm 5.16 multiplies states in a given family and returns a multiplied family. Each state of the original family is multiplied to n states, specified by a combination of pure incoming and pure outcoming transitions. If a state has a self-loop, then the whole self-loop is assigned to each new state. The original state is substituted by a set of new states, even if the state already applies to the condition of maximal one incoming and outcoming transition. The original state is removed after the multiplication. The returned family consists only of newly created states.

5.3 Approximation of Language Equivalence

The language equivalence calculation in a SAT-solver-based minimization is very hard because the reduction algorithm multiplies a family to many states (even hundreds of states). The calculation of the simulation relation is slow too. The multiplication is done many times in a minimization process and families are getting bigger and bigger. It is necessary to calculate only an under-approximation of language equivalence. The state equivalence checking algorithm with defined distance is an adaptation of an automata equivalence checking Algorithm 2.14. The approximation approach has a specified distance, on which the language equivalence of two states must be confirmed, otherwise, the states are not equivalent.

Algorithm 5.17 The approximation algorithm for forward language equivalence checking

```
    Input: NFA $M = (Q, \Sigma, \delta, I, F)$, two state $q, r \in Q$, and $distanceMax \in \mathbb{N}$
    Output: “Yes„ if $\overrightarrow{L}(q) \equiv \overrightarrow{L}(r)$, otherwise “No„
 1: $visited \leftarrow \emptyset$                ▷ contains allowable next states after reaching $distanceMax$
 2: $closed \leftarrow \emptyset$
 3: $todo \leftarrow \{(q, r)\}$
 4: $distance \leftarrow 0$
 5: while $todo \neq \emptyset$ do
 6:     Pick $(X, Y) \in todo$ and remove it
 7:     if $(X, Y) \notin closed$ then
 8:         $visited \leftarrow visited \cup X \cup Y$
 9:         if $X = Y$ then
10:             continue
11:         end if
12:         if $(X, Y)$ is bad pair then
13:             return “No, $\overrightarrow{L}(q) \neq \overrightarrow{L}(r)$„
14:         end if
15:         for $a \in \Sigma$ do
16:             if $distance = distanceMax$ then
17:                 if $\delta(X, a) \setminus visited \neq \emptyset \ \vee \ \delta(Y, a) \setminus visited \neq \emptyset$ then
18:                     return “No, $\overrightarrow{L}(q) \neq \overrightarrow{L}(r)$„
19:                 end if
20:             end if
21:             $todo \leftarrow todo \cup \{(\delta(X, a), \delta(Y, a))\}$
22:         end for
23:         $closed \leftarrow closed \cup \{(X, Y)\}$
24:         $distance \leftarrow distance + 1$
25:     end if
26: end while
27: return “Yes $\overrightarrow{L}(q) \equiv \overrightarrow{L}(r)$„
```

An empty set *closed* is initialized at the beginning of the algorithm. The *closed* set will store the already processed pairs and prevents the algorithm from an infinite loop.

Thereafter, the set *todo* will be initialized, and the pair $(\{p\}, \{q\})$ is inserted. The pair of a set of states (X, Y) is equivalent if X and Y are identical. The pair (X, Y) is bad if one of the sets is empty, but the other is not, or if a state of X or Y is final whereas the other is not. Otherwise, successors for X and Y are generated into *todo*. If the set *todo* is empty, then the languages of states are equivalent. After reaching a maximal destination, the calculation can continue, but only with states in *visited*. Any other state means that the language equivalence cannot be determined at the given distance. The algorithm for backward language equivalence checking uses a backward transition function δ^{-1} instead of δ.

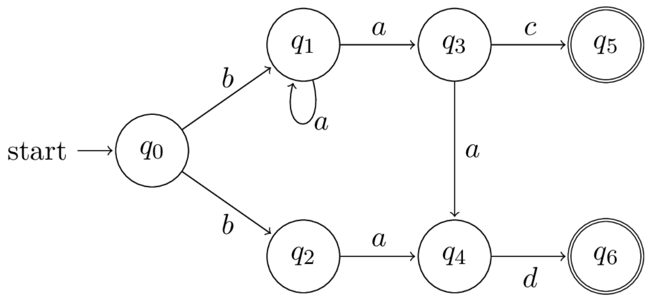


Figure 5.18: States q_1 and q_2 are backward equivalent at distance 1, but states q_3 and q_4 are not. Nevertheless, the states q_3 and q_4 are equivalent on a distance 2.

5.4 SAT Solver Coding

As has been already mentioned, the SAT solver gives information on how to perform the most optimal merging of states of the family. The main information for the solver coding is language equivalence. Only states with forward or backward language equivalence can be merged. The problem of optimal state merging is that some states can be in more than one language equivalence. For example, state q_1 is in backward language equivalence with a state q_2 and in forward language equivalence with state q_0. This example is shown in Figure 5.19.

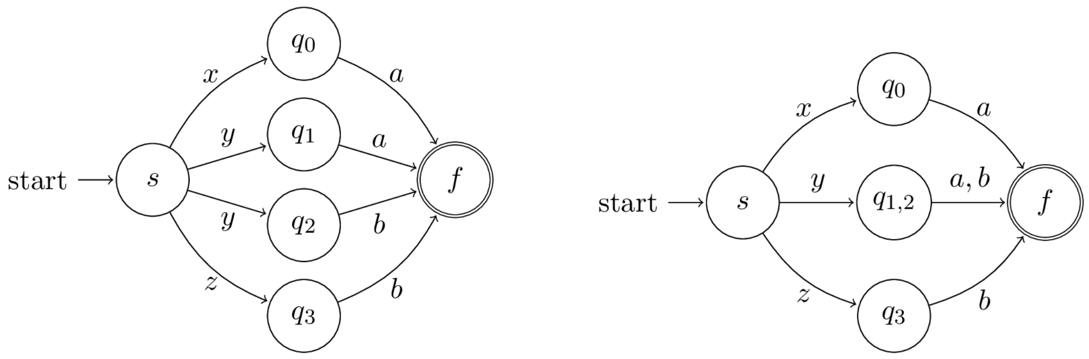


Figure 5.19: A suboptimal merge of an automaton on the left. States q_1 and q_2 were merged based on a backward language equivalence. This backward merge blocks a merge of states q_0, q_1 and q_2, q_3. based on the forward language equivalence.

The minimization cannot be optimal if the merge is not performed based on the SAT solver information. For example, if a state q_1 is merged with a state q_2 based on a backward language equivalence, then the state q_0 cannot be merged with a state q_1 based on a forward language equivalence. Every state can be merged based on *constant language equivalence information* (the information is not recalculated after merge) only with backward language equivalent states or only with forward language equivalent states, not with states from both groups. The main question is: Which states merge to get the most optimal solution? Let the SAT solver decide.

For the specification of SAT solver problem, it is necessary to define a set of pairs of states that can be merged based on the backward language equivalence and the set of pairs of states that can be merged based on the forward language equivalence.

Definition 5.20 *Let $\overleftarrow{Eq}$ be a set of pairs of states with equivalent backward language. The* ***set of backward equivalent pairs of states*** *is $\equiv_b = \{\{q, r\} \mid (q, r) \in \overleftarrow{Eq},\ q \neq r\}$*

Definition 5.21 *Let $\overrightarrow{Eq}$ be a set of pairs of states with equivalent forward language. The* ***set of forward equivalent pairs of states*** *is $\equiv_f = \{\{q, r\} \mid (q, r) \in \overrightarrow{Eq},\ q \neq r\}$*

The SAT solver calculates the most effective solution, which provides the maximum merged pairs of states, where each state must belong only to the backward equivalent pairs or to the forward equivalent pairs.

Definition 5.22 *The* ***problem of optimal merge*** *is a problem of maximizing the subsets $P \subseteq \equiv_f$ and $R \subseteq \equiv_b$, such that $\bigcup P \cap \bigcup R = \emptyset$.*

With definition of the solver problem and the definition of forward and backward equivalent pairs, the SAT solver coding of this problem can be described. Not only the variables and formulas will be defined in the following sections, but also their count. The size (count of formulas) of the solver problem is the main key for SAT solver running time. The next goal in the future will be to decrease these numbers and speed up SAT solver decision.

5.4.1 Variables

Maximal two boolean variables will be used for each state of the *subfamily* in a solver coding. The family is splitted by Algorithm 5.8. There is no language equivalence relation between two states from different subfamilies. Splitting will speed up solver calculation. A prefix B is used for the states in $\equiv_b$ and a prefix F is used for the states in $\equiv_f$. The value of the variable signifies if a state is merged based on forward or backward language equivalence.

Definition 5.23 *Let $\equiv_b$ be a set of backward equivalent pairs of states and $\equiv_f$ be a set of forward equivalent pairs of states. Then the* ***solver variables*** *are defined as $Vars = \{Bq \mid q \in \bigcup \equiv_b\} \cup \{Fq \mid q \in \bigcup \equiv_f\}$.*

Let a subfamily has a set of backward language equivalent pairs of states $\equiv_b = \{(q_0, q_1), (q_2, q_3)\}$ and forward language equivalent pairs $\equiv_f = \{(q_0, q_3), (q_4, q_5), (q_2, q_5)\}$. Then the variables are Fq_0, Fq_1, Fq_2, Fq_3, Bq_0, Bq_2, Bq_3, Bq_4 and Bq_5.

One state can be merged only based on the backward language equivalence or based on the forward language equivalence. Therefore, the maximal number of variables used in the SAT solver coding will be twice the number of states.

Lemma 5.24 *Let the family has n members. The number of solver variables is maximal:*

$$Vars_{cnt}(n) = 2n \tag{5.25}$$

5.4.2 Merge Formula

The language equivalence (backward or forward) and thus the merge of two states based on the language equivalence (backward or forward) is coded by logical conjunction. Two states can be merged only if both these states allow the merge. The states p and q with backward language equivalence can be merged only if the state p and q are merged with some other state based on the backward language equivalence. That means that the formula $Bp \wedge Bq$ must be truly evaluated. On the other hand, only some pairs of states in backward and some pairs of states in forward language equivalent pairs of states will be merged. Not all logical conjunctions that symbolize state merge will be evaluated by true. The SAT solver task is to maximize the count of truly evaluated conjunctions. For this reason, the logical conjunctions are connected with logical disjunction.

Definition 5.26 *Let $\equiv_b$ be a set of backward equivalent pairs of states and $\equiv_f$ be a set of forward equivalent pairs of states. Then the* ***merge formula*** *is defined as:*

$$\bigvee \phi_{Merge} \tag{5.27}$$

where $\phi_{Merge} = \{Bq \wedge Br \mid (q, r) \in \equiv_b\} \cup \{Fq \wedge Fr \mid (q, r) \in \equiv_f\}$.

Let the set of backward language equivalent paris be $\equiv_b = \{(q_0, q_1), (q_2, q_3)\}$ and the set of forward language equivalent paris be $\equiv_f = \{(q_0, q_3), (q_4, q_5), (q_2, q_5)\}$. Then the merge formula is $(Bq_0 \wedge Bq_1) \vee (Bq_2 \wedge Bq_3) \vee (Fq_0 \wedge Fq_3) \vee (Fq_4 \wedge Fq_5) \vee (Fq_2 \wedge Fq_5)$.

Because in the worst case, all states can be backward and forward language equivalent, the maximal count of logical conjunctions (that code merge of two states) will be the combination of backward and forward language equivalent pairs of states.

Lemma 5.28 *Let the family has n members. The maximal count of a logical conjunction in the merge formula is:*

$$Conjunction_{cnt}(n) = 2 \cdot \binom{n}{2} = 2 \cdot \frac{n!}{(n-2)! \cdot 2!} = n \cdot (n-1) \tag{5.29}$$

5.4.3 Merge Rules

The most important part of the information in SAT solver coding are the rules, which block the merge of the state based on a forward language equivalence after performing the merge based on a backward equivalence, and vice versa. The rule must be created for states that are in both forward and backward language equivalence.

Definition 5.30 *Let $\equiv_b$ be a set of backward equivalent pairs of states and $\equiv_f$ be a set of forward equivalent pairs of states. Then the **merge rules formula** is defined as:*

$$\bigwedge \phi_{Rules} \tag{5.31}$$

where $\phi_{Rules} = \{Bq \implies \neg Fq \,|\, \{q, r\} \in \equiv_b \wedge \{q, s\} \in \equiv_f,$ *where* $q, r, s \in Q\}$.

Let the set of backward language equivalent paris be $\equiv_b = \{(q_0, q_1), (q_2, q_3)\}$ and the set of a forward language equivalent paris be $\equiv_f = \{(q_0, q_3), (q_4, q_5), (q_2, q_5)\}$. Then the merge rule formula is $(Bq_0 \implies \neg Fq_0) \wedge (Bq_2 \implies \neg Fq_2) \wedge (Bq_3 \implies \neg Fq_3)$.

In the worst case, all states will be used in the forward and backward language equivalent pairs of states. That means that the rule will be created for every state.

Lemma 5.32 *Let the family has n members. The maximal count of implications in a merge rule formula is:*

$$Implication_{cnt}(n) = n \tag{5.33}$$

5.4.4 Coding

With the information, such as solver variables, merge formulas, and rules, the merge problem can be coded.

Definition 5.34 *The **minimization formula** is defined as a logical confunction of merge formula and merge rules formula:*

$$MergeFormula \;\wedge\; MergeRulesFormula. \tag{5.35}$$

Let the set of backward language equivalent paris be $\equiv_b = \{(q_0, q_1), (q_2, q_3)\}$ and the set of forward language equivalent paris be $\equiv_f = \{(q_0, q_3), (q_4, q_5), (q_2, q_5)\}$. The the minimization formula is:

$$\begin{aligned}&\big((Bq_0 \wedge Bq_1) \vee (Bq_2 \wedge Bq_3) \vee (Fq_0 \wedge Fq_3) \vee (Fq_4 \wedge Fq_5) \vee (Fq_2 \wedge Fq_5)\big)\wedge \\ &\big((Bq_0 \implies \neg Fq_0) \wedge (Bq_2 \implies \neg Fq_2) \wedge (Bq_3 \implies \neg Fq_3)\big)\end{aligned} \tag{5.36}$$

We use SAT solver Z3[2] for solving the problem of optimal merge in this thesis. The problem of optimal merge is written in the Z3 solver syntax (lower case prefixes b and f are used due to Z3 syntax rules) as:

```
; variables
(declare-const bq_0 Bool)
(declare-const bq_1 Bool)
(declare-const bq_2 Bool)
(declare-const bq_3 Bool)
(declare-const fq_0 Bool)
(declare-const fq_2 Bool)
(declare-const fq_3 Bool)
(declare-const fq_4 Bool)
```

2 Z3 solver is available at https://github.com/Z3Prover/z3

```
(declare-const fq_5 Bool)
; rules
(assert (=> bq_0 (not fq_0)))
(assert (=> bq_2 (not fq_2)))
(assert (=> bq_3 (not fq_3)))
; merge formulas
(assert-soft (and bq_0 bq_1) :weight 1)
(assert-soft (and bq_2 bq_3) :weight 1)
(assert-soft (and fq_0 fq_3) :weight 1)
(assert-soft (and fq_2 fq_5) :weight 1)
; run the solver and get the result
(check-sat)
(get-model)
```

The solver is forced to maximize a truly evaluated conjunctions in the merge formula [7]. Z3 solver returns all variables and its evaluation. Only pairs of states where both states were assigned to true, will be merged together. In our example, the solver evaluates by true variables Bq_0, Bq_1, Fq_2, and Fq_5. With the information about language equivalent pairs, it can be seen that only pairs (q_0, q_1) based on a backward language equivalence and (q_2, q_5) based on a forward language equivalence will be merged.

5.5 SAT-solver-based Minimization

This section describes an automaton minimization process with the usage of the techniques described in this chapter, such as finding families, multiplying, and solver problem coding.

Equivalent initial or final states cannot be sometimes minimized, because the algorithm minimizes only families. And the family consists of states with a common ancestor or successor. Initial states do not have any ancestor. For the same reason, the two final states without forward transition that are forward equivalent cannot be in the family. It is necessary to modify an automaton to an automaton with only one initial and maximal two final states (one can be final and initial).

5.5.1 One Initial State

If the automaton has more than one initial state, then this initial behavior can be replaced by a creation of a new initial state. All transitions going from the old initial states will be duplicated and assigned to the new initial state.

Definition 5.37 *Let $M = (Q, \Sigma, \delta, I, F)$ be the NFA. The* ***automaton with one initial state*** *is $M_I = (Q_I, \Sigma, \delta_I, q_0, F)$ where:*

1. *$Q_I = Q \cup \{q_0\}$ is a finite set of states,*
2. *Σ is an alphabet,*
3. $\delta_I(s, a) = \begin{cases} \bigcup_{p \in I} \delta(p, a), & \textit{for } s = q_0, \\ \delta(s, a), & \textit{otherwise.} \end{cases}$
4. *$I = \{q_0\}$ is a set containing one initial state, and*

5. *$F \subseteq Q$ is a set of final states.*

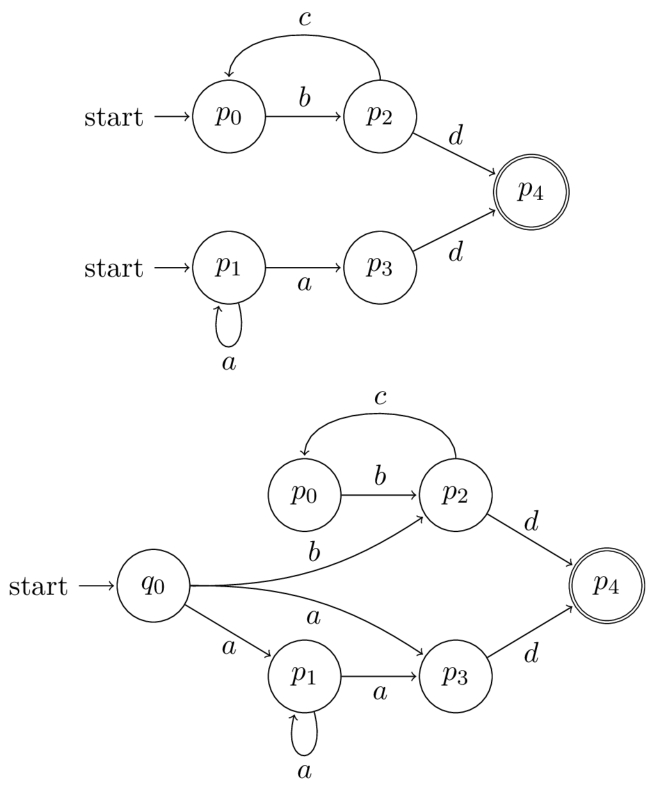


Figure 5.38: The automaton M on the top with two initial states (p_0 and p_1) and the equivalent automaton M_I with one initial state (q_0) at the bottom.

5.5.2 Central Final State

If an automaton has more than one final state, then the final behavior of these states can be replaced by a creation of the new final state. And all transitions going to the old final states will be duplicated and assigned to the new final state. All old states that are not initial will lose their final behavior. All states that are final and initial remain final because it is not always easy to create the automaton containing only one initial and one final state.

Definition 5.39 *Let $M = (Q, \Sigma, \delta, I, F)$ be NFA. The* ***automaton with one final state*** *is $M_F = (Q_F, \Sigma, \delta_F, I, F_F)$ where:*

1. *$Q_F = Q \cup \{f_0\}$ is a finite set of states,*

2. *Σ is an alphabet,*

3. $\delta_F(s, a) = \begin{cases} \bigcup_{p \in F} \delta(p, a) & \text{for } s = f_0, \\ \delta(s, a) & \text{otherwise.} \end{cases}$

4. *$I \subseteq Q$ is a initial state, and*

5. *$F_F = (I \cap F) \cup \{f_0\}$ is a set of final states.*

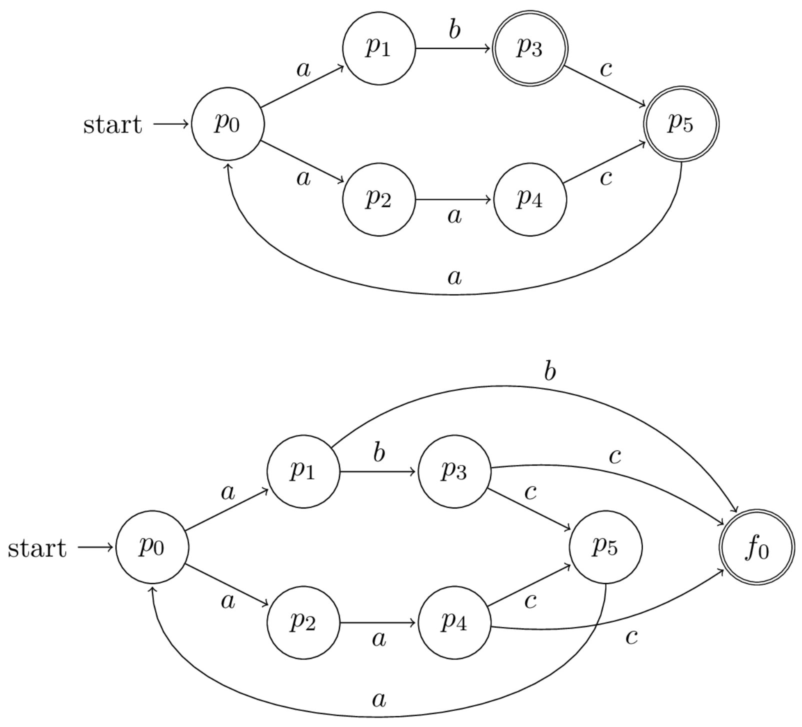


Figure 5.40: The automaton M on the top with two final states (p_3 and p_5) and at the bottom equivalent automaton M_F with one final state (f_0).

5.5.3 SAT-solver-based Minimization Algorithm

At the beginning of the SAT-solver-based minimization algorithm, the *closed* set is initialized to $\emptyset$. This set will contain already minimized families. The input automaton is modified to have only one initial and one central final state. The families are found by Algorithm 5.11. The automaton is reduced while there exist such $families$ that are not yet in the *closed* set. Each $family$ is backuped. The *backup* contains states, their initial or final behavior, and transitions of states. The *backup* will be used if the SAT solver returns the bigger family than the family was. The family is multiplied by Algorithm 5.16. The multiplication is done only once before merge. While it is possible to merge some states in the family, the SAT solver returns the most optimal merge suggestion. The family is then merged according to this suggestion. After the SAT solver reduction, the family is added to the *close* set. If all families are already in *close* set, then the result automaton is returned.

Algorithm 5.41 SAT-solver-base minimization algorithm of NFA

```
Input: NFA $M = (Q, \Sigma, \delta, I, F)$
 1: $closed \leftarrow \emptyset$
 2: $M \leftarrow M.makeOneInitialState()$
 3: $M \leftarrow M.makeCentralFinalState()$
 4: while True do
 5:     $families \leftarrow M.findFamilies()$
 6:     $families \leftarrow families \setminus closed$
 7:     if $families = \emptyset$ then
 8:         break
 9:     end if
10:     for $family \in families$ do
11:         $backup \leftarrow$ create $backup$ of the states and transitions of the $family$
12:         $family \leftarrow multiplyStates(family)$
13:         while True do
14:             $mergeSuggestion \leftarrow SATsolverMergePrediction(family)$
15:             if $mergeSuggestion = \emptyset$ then
16:                 break
17:             end if
18:             for $groupOfStates \in joinSetsWithCommonElem(mergeSuggestion)$ do
19:                 $family \leftarrow family \cup mergeStates(groupOfStates)$
20:                 $family \leftarrow family \setminus groupOfStates$
21:             end for
22:         end while
23:         if $family > backup$ then
24:             for $state \in family$ do
25:                 remove $state$ and its transitions from $M$
26:             end for
27:             for $state \in backup$ do
28:                 restore states and transitions from $backup$
29:             end for
30:             $family \leftarrow$ get states from $backup$
31:         end if
32:         $closed \leftarrow closed \cup \{family\}$
33:     end for
34: end while
```

Chapter 6

Experiments

We test the efficiency of SAT-solver-based reduction algorithm using Z3 solver on two sets of automata and compare it with the existing tool RABIT/Reduce (version 2.5) [12] running on Java 11.0.11. The fist set was created from regular expressions, particular from databases of network intrusion detection systems Bro [5] and Snort [4], the academic papers [11, 34], the RegExLib database [29], and industrial regexes [21] used for security purpose. The second set containing bigger (with more transitions) automata was constructed from Nested antichains for WS1S [16]. The SAT-solver-based reduction algorithm was implemented with Python 3.8.5. The experiments were performed on one thread of AMD Ryzen 7 3800XT 8-Core and 32 GB of memory.

The chapter also provides a comparison of SAT solver running time in dependence on the size of the automaton alphabet and a size of a family of states, on the most difficult type of family (1:1).

6.1 Reduction of Regexes

The efficiency of the investigated approach is compared with a tool RABIT/Reduce on 3730 automata, with a total of 63538 states and 84093 transitions. Automata were modified to have one initial and maximal two final states (one can be final as well as initial). The size of automata is between 10 and 400 states. The average count of transitions per state is 1.57 (from each state leads approximately 1.57 transitions). The bigger the transition density is, the slower the solver minimization is. First, the solver is compared with RABIT/Reduce, which uses state merging and transition pruning. Then the solver is used as a supplement of RABIT/Reduce after running a merge, transition pruning, and saturation, which is the best-known combination. RABIT/Reduce uses lookahead simulation for an approximation of language relations. Lookahead was set to 1 for all experiments. Bigger lookahead did not give better minimization results, only slower down the RABIT/Reduce.

6.1.1 SAT Solver vs RABIT

The new minimization approach using SAT solver information for better state merging and maximal distance 10 in the algorithm for checking of the language equivalence 5.17 minimizes the input automata to a total of 59421 states. The tool RABIT/Reduce using state merging and transition pruning minimizes the input automata to a total of 58734 states. This means that the minimization using the SAT solver approximated the RABIT/Reduce result with an accuracy of 98.84%.

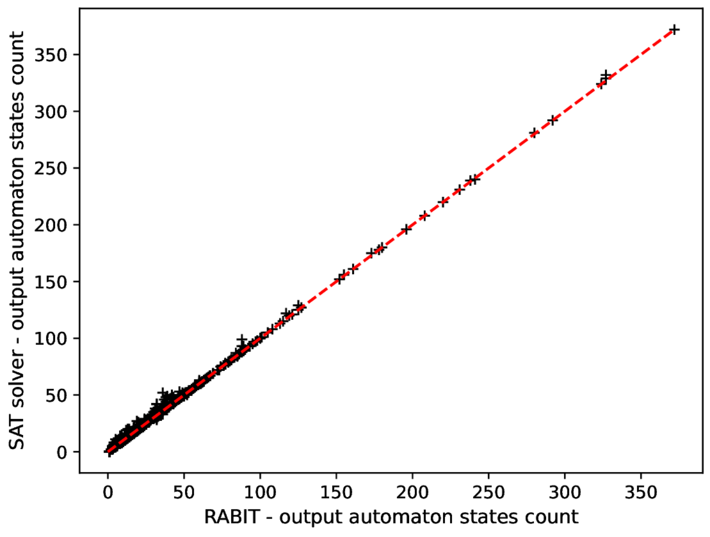


Figure 6.1: Comparison of the count of states of the results of SAT-solver-based reduction algorithm with maximal distance 10 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning. SAT-solver-based reduction approximates RABIT/Reduce result at 98.84%.

In this set of states, the maximal distance 1 has the minimal effect on the time of the calculation of language equivalence (42.340 sec with maximal distance 10 and 40.700 sec with maximal distance 1), but gives much worse results (59421 states with maximal distance 10 and 60313 with maximal distance 1).

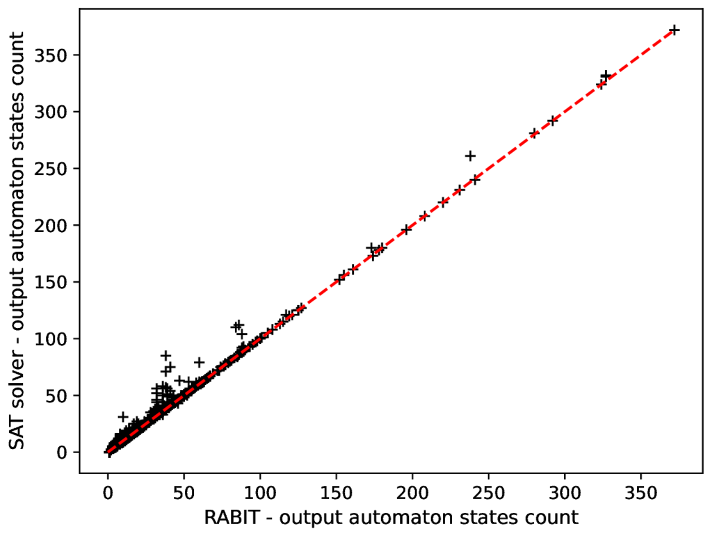


Figure 6.2: Comparison of the count of states of the results of SAT-solver-based reduction algorithm with maximal distance 1 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning.

The SAT-solver-based algorithm maximized states reduction, not transition reduction. SAT-solver-based minimization reduced the input automata to a total of 77945 transitions. Nevertheless, RABIT/Reduce returns automata with 75316 transitions. The comparison is in Figure 6.3

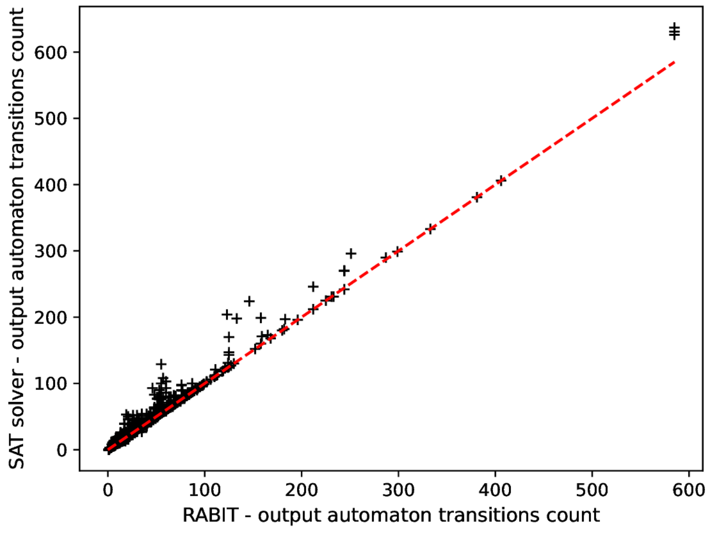


Figure 6.3: Comparison of the count of transitions of the results of SAT-solver-based reduction algorithm with maximal distance 10 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning.

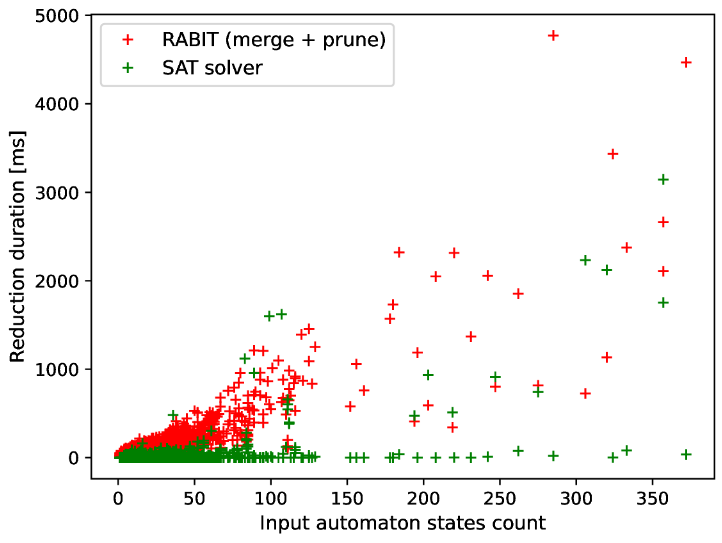


Figure 6.4: Comparison of the reduction time of the SAT-solver-based reduction algorithm with maximal distance 10 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning. The SAT-solver-based algorithm reduces automata 7.5 times faster than the tool RABIT/Reduce.

Another engaging part of the comparison is the minimization duration. The RABIT/Reduce, using state merging and transition pruning, minimizes 3730 automata in 317.830 s. On the contrary, the solver did approximately the same minimization in only 42.340 s. That is 13.32% of the time consumed by RABIT/Reduce. The comparison is in Figure 6.4. However, the solver minimization is slower on automata with high transition density.

6.1.2 SAT Solver as RABIT's Supplement

The solver with merge can be used alone or as a supplement of a RABIT/Reduce. The solver with merge is processed over a RABIT/Reduce best minimization. The new reduced automaton is a little more minimal than the RABIT/Reduce result. The difference between the minimized states (the difference between a number of states of the input and output automaton) of RABIT/Reduce itself and its version enriched by the solver is the main measurement. The total input number of minimized states is the same as in the previous example (63538). RABIT/Reduce itself minimizes the automata by 5322 states. The extended version of the algorithm using RABIT and solver minimizes the automata by 5356 states. That is, about 0.63% minimized states more than only by using RABIT/Reduce, which uses the strongest minimization algorithms.

6.2 Reduction of WS1S

The second group contains 1132 automata with a total of 35389 states and 518342 transitions (14.6 transitions per state). Only the SAT-solver-based reduction algorithm with maximal distance 1 and RABIT/Reduce with lookahead 1 using state merging and transition pruning were compared, because these automata do not contain any fragments than RABIT/Reduce itself can reduce. This time, the SAT-solver-based reduction gives a better reduction of states and transitions. However, the reduction time was higher than RABIT/Reduce due to the big size of minimized families (caused by many transitions) as shown in the figure 6.8

6.2.1 SAT Solver vs RABIT

The SAT-solver-based reduction approach minimized the input automata to a total of 20340 states and 119103 transitions. The tool RABIT/Reduce using state merging and transition pruning minimizes the input automata to a total of 21236 states and 126237 transitions. That means that the solver reduced automata about 896 (6.33%) states and 7134 (1.82%) transitions more than RABIT/Reduce. The comparison is in Figure 6.5 and 6.6. Unfortunately, the RABIT/Reduce was on average 5.5 times faster than SAT-solver-based algorithm. High computation time of SAT-solver-based reduction was caused due to the big size of families, which slow down the solver calculations, as shown in Figure 6.8.

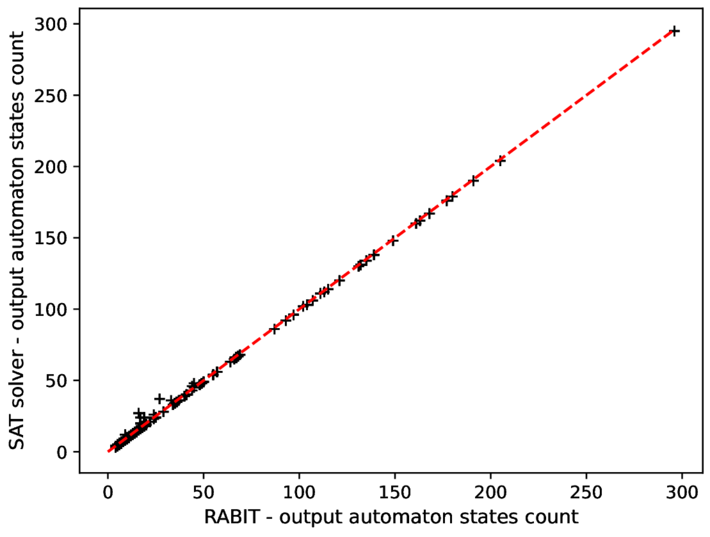


Figure 6.5: Comparison of the number of states of the results of SAT-solver-based reduction algorithm with maximal distance 1 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning. SAT-solver-based algorithm reduced about 896 (6.33%) states more.

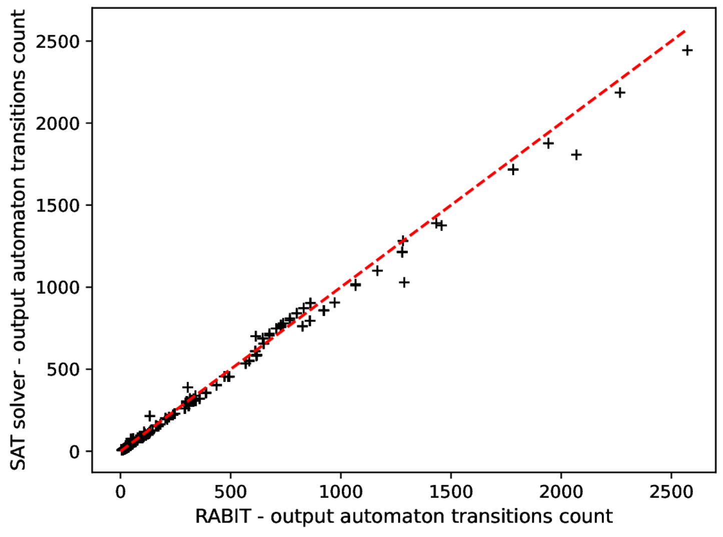


Figure 6.6: Comparison of the number of transitions of the results of SAT-solver-based reduction algorithm with maximal distance 1 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning. SAT-solver-based algorithm reduced about 7134 (1.82%) transitions more.

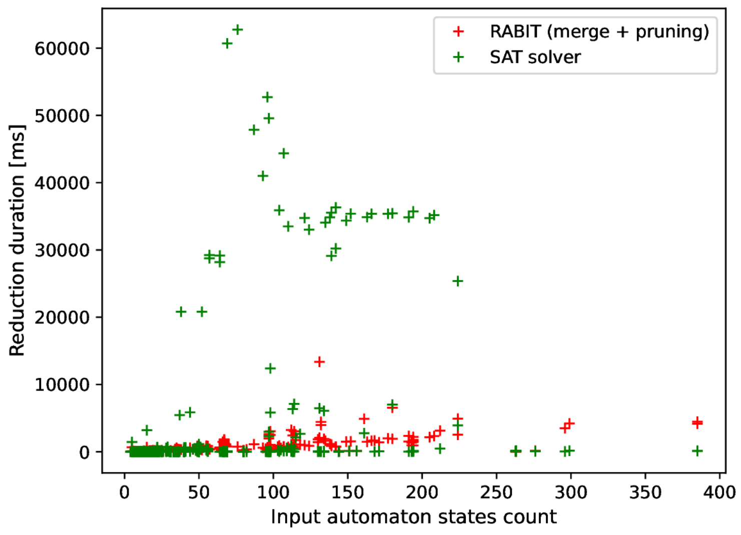


Figure 6.7: Comparison of the reduction time of the SAT-solver-based algorithm with maximal distance 1 and RABIT/Reduce with lookahead 1, which uses state merging and transition pruning.

SAT-solver-based algorithm reduced automata slower than RABIT in this example. SAT-solver-based algorithm with maximal distance 1 minimized 1132 automata in 1335.461 sec, but RABIT with lookahead 1 minimized automata in 241.727 sec. The time difference was caused by a high size of minimized families as shown in Figure 6.9

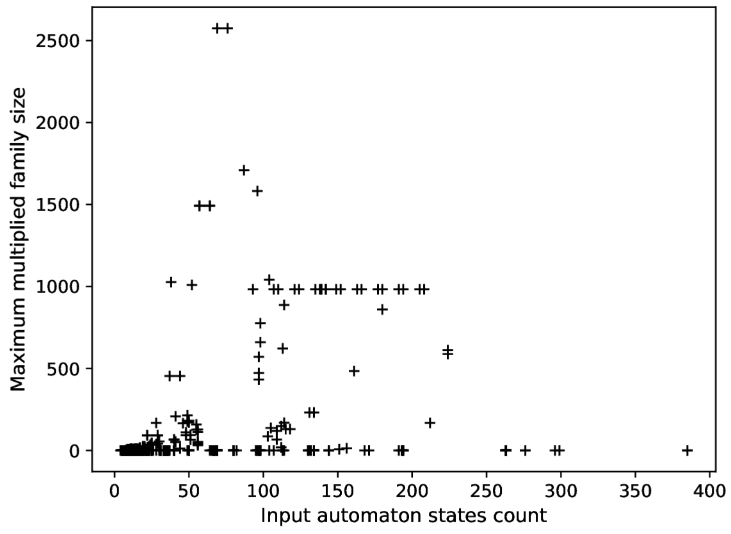


Figure 6.8: Dependancy of input automaton size and maximal size of minimized family during the reduction.

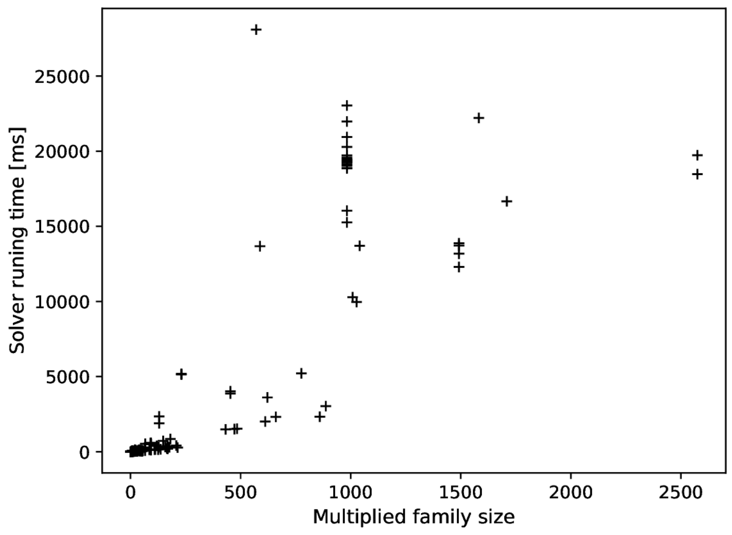


Figure 6.9: Experimental values of SAT solver Z3 running time in dependence on a maximal minimized family size.

6.3 Reduction Time Based on the Family Size

This section displays the impact of family size to the time consumed by a SAT solver Z3. The solver running time increases with the number of language equivalent states in the family. The graphs below show the worst reduction scenario on family 1:1.

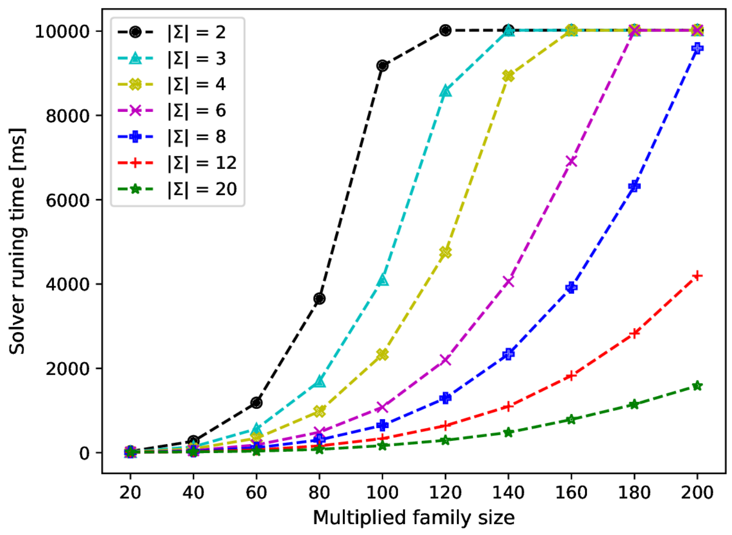


Figure 6.10: Comparison of the SAT solver Z3 running time in dependence on the size of the multiplied family and the size of alphabet. Each point is the average of 100 random families 1:1. SAT solver was finished after reaching 10 seconds.

The tests were performed on automata consisting of only one random family 1:1 between 20 and 200 states. All family states have only one incoming and one outcoming transition, with no self-loops. The size of the alphabet is between 2 and 20 characters. This setup represents the worst reduction scenario because each state can be a backward and forward language equivalent to any other. Small alphabet size allows the existence of many equivalent states and thus extremely slows down the SAT solver Z3 decision.

The SAT solver running time depends on the number of conjunctions in the merge formula (language equivalent pairs of states). If the letters of the alphabet are uniformly assigned to the family transitions and n is a count of the family states, then the number of conjunctions in the merged formula 5.26 is:

$$Conjunction_{cnt}(n, |\Sigma|) = \frac{2}{|\Sigma|} \cdot \binom{n}{2} = \frac{2}{|\Sigma|} \cdot \frac{n!}{(n-2)! \cdot 2!} = \frac{n \cdot (n-1)}{|\Sigma|} \tag{6.11}$$

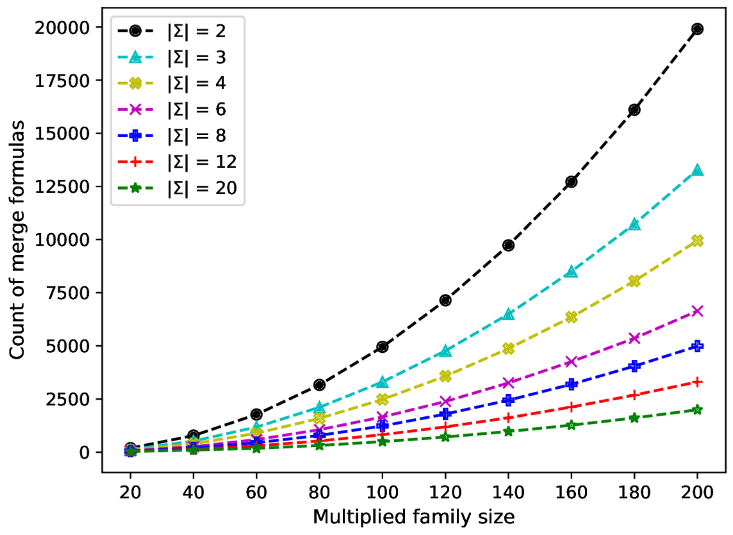


Figure 6.12: Dependance between the size of multiplied family 1:1, alphabet and conjunctions in the merge formula 5.26. Each point is the average of 100 random families 1:1.

It could be seen that the SAT-solver-based reduction algorithm works faster for the families with small count of language equivalent states. Based on that and the previous experiments, we can say that the SAT-solver-based reduction algorithm works best for the automata containing maximal three times more transitions than states.

The explosion of the count of conjunction in the merge formula can be solved using sets in the future. For the family of 200 states and alphabet of size 2 we do not need 19900 conjunctions, but only a set of backward language equivalent states using letter a, the set of backward language equivalent states using letter b, and two same sets for forward equivalent states. That is only 4 sets containing 200 states together.

Chapter 7

Conclusion

Nondeterministic finite automata can choose from more than one transition after receiving the letter. This feature allows NFA to represent the language with fewer states and transitions than its deterministic variant but makes the reduction difficult.

Nowadays, many efficient reduction algorithms exist, such as state merging, transition pruning, and saturation. All these techniques are implemented in the reduction tool RABIT/Reduce to which we compared our investigated SAT-solver-based reduction algorithm.

The SAT-solver-based method focuses on the local reduction of the sets of states with common ancestors or common successors, where part of the language of some state can be already covered by the languages of the other states. These groups cannot be sometimes reduced by the existing algorithms. SAT-solver-based algorithm simplifies (by adding states and transitions) these groups of states so that every state of the group has maximal one incoming and one outcoming transition, without changing the automaton language. This group is then merged. The SAT solver is used to maximize the number of merged pairs of states.

The investigated reduction algorithm strongly approximates (at 98.84%) the reduction results of the tool RABIT/Reduce which uses state merge and transition pruning. The SAT-solver-based algorithm gives 7.5 times faster results on automata with 1.3 transitions per state. This algorithm is suitable to use for automata which have maximal 3 times more transitions than states. For denser automata, the result still strongly approximates the RABIT/Reduce solution, but slower.

In the future, we would improve the coding of the problem of optimal merge for faster SAT solver calculation. As has been already mentioned, the minimization using a solver works slower for dense automata which have more than 3 transitions per state. The method could be improved to work better for automata with dense transitions. Due to a high and fast approximation of RABIT/Reduce merge and transition pruning results, the solver minimization could replace the merging and transition pruning phase in standard minimization algorithms.

Bibliography

[1] Abdulla, A. P., Holík, L., Chen, Y.-F., Mayr, R. and Vojnar, T. *When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata)*. 2010. 22 p. Available at: https://www.fit.vut.cz/research/publication/9152.

[2] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Holík, L., Rezine, A. et al. String Constraints for Verification. In: Biere, A. and Bloem, R., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2014, p. 150–166. ISBN 978-3-319-08867-9.

[3] Ain, Q., Saeed, Y., Naseem, S., Ahamd, F., Alyas, T. et al. DNA Pattern Analysis using Finite Automata. *International Research Journal of Computer Science (IRJCS)*. october 2014, vol. 1, p. 1–4.

[4] al., M. R. et. *Snort: A Network Intrusion Detection and Prevention System*. Available at: https://www.snort.org/.

[5] al., R. S. et. *The Bro Network Security Monitor*. Available at: http://www.bro.org.

[6] Aziz, A., Singhal, V., Brayton, R. and Swamy, G. Minimizing interacting finite state machines: a compositional approach to language containment. In: *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 1994, p. 255–261. DOI: 10.1109/ICCD.1994.331900.

[7] Bjørner, N. and Dung, P. VZ - Maximal Satisfaction with Z3. In: *Proceedings of the 6th International Symposium on Symbolic Computation in Software Science (SCSS 2014)*. 2015. 6th International Symposium on Symbolic Computation in Software Science (SCSS 2014), SCSS ; Conference date: 07-12-2014 Through 08-12-2014. Available at: http://www.easychair.org/smart-program/SCSS2014/index.html.

[8] Bonchi, F. and Pous, D. *Hopcroft and Karp's algorithm for Non-deterministic Finite Automata*. November 2011. 26p. Available at: https://hal.archives-ouvertes.fr/hal-00639716.

[9] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*. Apr 2012, vol. 14, no. 2, p. 167–191. DOI: 10.1007/s10009-011-0205-y. ISSN 1433-2787. Available at: https://doi.org/10.1007/s10009-011-0205-y.

[10] Bustan, D. and Grumberg, O. Simulation Based Minimization. In: McAllester, D., ed. *Automated Deduction - CADE-17*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 255–270. ISBN 978-3-540-45101-3.

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

[12] Chen, Y.-F. and Mayr, R. *RABIT/Reduce: Tools for language inclusion testing and reduction of nondeterministic Büchi automata and NFA*. Available at: http://languageinclusion.org/doku.php?id=tools.

[13] Clemente, L. and Mayr, R. Efficient reduction of nondeterministic automata with application to language inclusion testing. *CoRR*. 2017, abs/1711.09946. Available at: http://arxiv.org/abs/1711.09946.

[14] Dill, D. L., Hu, A. J. and Wong Toi, H. Checking for language inclusion using simulation preorders. In: Larsen, K. G. and Skou, A., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, p. 255–265. ISBN 978-3-540-46763-2.

[15] Elgaard, J., Klarlund, N. and Møller, A. MONA 1.x: New techniques for WS1S and WS2S. In: Hu, A. J. and Vardi, M. Y., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, p. 516–520. ISBN 978-3-540-69339-0.

[16] Fiedor, T., Holík, L., Lengál, O. and Vojnar, T. Nested antichains for WS1S. *Acta Informatica*. 2019, vol. 56, no. 3, p. 205–228. DOI: 10.1007/s00236-018-0331-z. Available at: https://doi.org/10.1007/s00236-018-0331-z.

[17] Fu, C., Deng, Y., Jansen, D. and Zhang, L. On Equivalence Checking of Nondeterministic Finite Automata. In: Larsen, K. G., Sokolsky, O. and Wang, J., ed. *Dependable Software Engineering. Theories, Tools, and Applications*. Cham: Springer International Publishing, 2017, p. 216–231. ISBN 978-3-319-69483-2.

[18] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J. and Vojnar, T. Forest automata for verification of heap manipulation. *Formal Methods in System Design*. Aug 2012, vol. 41, no. 1, p. 83–106. DOI: 10.1007/s10703-012-0150-8. ISSN 1572-8102. Available at: https://doi.org/10.1007/s10703-012-0150-8.

[19] Heizmann, M., Hoenicke, J. and Podelski, A. Software Model Checking for People Who Love Automata. In: Sharygina, N. and Veith, H., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 36–52. ISBN 978-3-642-39799-8.

[20] Henzinger, M. R., Henzinger, T. A. and Kopke, P. W. Computing Simulations on Finite and Infinite Graphs. In: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. USA: IEEE Computer Society, 1995, p. 453–462. FOCS '95. ISBN 0818671831.

[21] Holík, L., Lengál, O., Saarikivi, O., Turoňová, L., Veanes, M. et al. Succinct Determinisation of Counting Automata via Sphere Construction. In: Lin, A. W., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2019, p. 468–489. ISBN 978-3-030-34175-6.

[22] Ilie, L., Navarro, G. and Yu, S. On NFA Reductions. In: Karhumäki, J., Maurer, H., Păun, G. and Rozenberg, G., ed. *Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 112–124. DOI: 10.1007/978-3-540-27812-2_11. ISBN 978-3-540-27812-2. Available at: https://doi.org/10.1007/978-3-540-27812-2_11.

[23] Ilie, L. and Yu, S. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*. 2003, vol. 306, no. 1, p. 373–390. DOI: https://doi.org/10.1016/S0304-3975(03)00311-6. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/S0304397503003116.

[24] Iliea, L. and Yu, S. Algorithms for Computing Small NFAs. In: Diks, K. and Rytter, W., ed. *Mathematical Foundations of Computer Science 2002*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, p. 328–340. ISBN 978-3-540-45687-2.

[25] Kim, H. and Choi, K.-I. A Pipelined Non-Deterministic Finite Automaton-Based String Matching Scheme Using Merged State Transitions in an FPGA. *PloS one*. United States: Public Library of Science. 2016, vol. 11, no. 10, p. 1–24. ISSN 1932-6203. Available at: https://doi.org/10.1371/journal.pone.0163535.

[26] Meduna, A. and Lukáš, R. *Formal Languages and Compilers: Models for regular languages* [lecture notes for Formal Languages and Compilers]. Božetěchova 1/2, 612 00 Brno-Královo Pole: FIT Brno University of Technology, 2017.

[27] Norton, D. Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP. april 2009.

[28] Rabin, M. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*. april 1959, vol. 3, p. 114–125. DOI: 10.1147/rd.32.0114.

[29] RegExLib.com.. *The Internet's first Regular Expression Library*. Https://regexlib.com/.

[30] Sidhu, R. and Prasanna, V. Fast Regular Expression Matching Using FPGAs. In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, February 2001, p. 227 – 238. ISBN 0-7695-2667-5.

[31] Sipser, M. Nondeterminism. In: *Introduction to the Theory of Computation*. 2nd ed. Boston: Thomson Course Technology, 2006. ISBN 0-534-95097-3.

[32] Sourdis, I. and Pnevmatikatos, D. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In: Y. K. Cheung, P. and Constantinides, G. A., ed. *Field Programmable Logic and Application*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, p. 880–889. ISBN 978-3-540-45234-8.

[33] Wolper, P. and Boigelot, B. On the Construction of Automata from Linear Arithmetic Constraints. In: Graf, S. and Schwartzbach, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 1–19. ISBN 978-3-540-46419-8.

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