BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REPETITIVE SUBSTRUCTURES FOR EFFICIENT REPRESENTATION OF AUTOMATA

VYUŽITÍ OPAKUJÍCÍCH SE PODSTRUKTUR PRO EFEKTIVNÍ REPREZENTACI AUTOMATŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL ŠEDÝ**
AUTOR PRÁCE

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

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

155613

Institut:	Department of Intelligent Systems (DITS)
Student:	**Šedý Michal, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Repetitive Substructures for Efficient Representation of Automata**
Category:	Algorithms and Data Structures
Academic year:	2023/24

Assignment:

The practical use of finite automata , whether in verification, in hardware, or in pattern search, requires efficient techniques for minimizing the size of automata and representing them efficiently.

One unexplored technique is to search for recurrent substructures in the graph of an automaton and represent the repeating instances by reference (similar to procedures in a program).

In this work, we will perform initial experiments with this idea. The will require completion of the following points:

1. Propose a theoretical model of an automaton with subprocedures.
2. Outline an algorithm for finding substructures.
3. Implement the algorithm.
4. Evaluate effectiveness of the technique (computational resources and compatness of the result).

Literature:

1. Javier Esparza. Automata Theory, The Algorithmic Approach. 2017. https://www7.in.tum.de/~esparza/automatanotes.html

Requirements for the semestral defence:
1, 2

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

Supervisor:	**Holík Lukáš, doc. Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

Nondeterministic finite automata (NFAs) are widely used across almost every field of computer science, such as for the representation of regular expressions, monitoring high-speed networks, in abstract regular model checking, program verification, in decision procedures of WS1S and WS2S logics, linear integer arithmetic, temporal logics, or even in bioinformatics for searching sequences of nucleotides in DNA. Automata with a large number of states can lead to an exponential increase in the state space in many algorithms. To address this issue, minimization techniques, such as state merging and transition pruning, are used. Despite the strong minimization potential of these methods, the resulting automata can still contain duplicate substructures with equivalent transition sequences. There are even types of automata that cannot be minimized by these standard methods at all. This work presents a novel automata minimization approach based on a transformation of an NFA into a nondeterministic pushdown automaton (NPDA). The transformation identifies multiple similar substructures and replaces them with one common structure (called a procedure). By doing so, we were able to further reduce automata by up to 67.3%. The principle of transforming NFA into NPDA can be understood as a transformation of a purely sequential program into a program with functions and a call stack.

Abstrakt

Nedeterministické konečné automaty (NKA) jsou široce využívány napříč mnoha odvětvími počítačové vědy, například pro reprezentaci regulárních výrazů, při monitorování vysokorychlostních sítí, v abstraktním regulárním model checkingu, k verifikaci programů, k rozhodování procedur logik WS1S a WS2S, lineární aritmetiky celých čísel, temporálních logik, nebo dokonce v bioinformatice při vyhledávání sekvencí nukleotidů v DNA. Automaty s velkým množstvím stavů mohou v řadě algoritmů vést k exponenciálnímu nárůstu stavového prostoru. Tento problém lze zmírnit použitím minimalizačních technik slučování stavů a prořezávání hran přechodů. Tyto metody však mohou i přes svou značnou efektivitu zanechat ve výsledných automatech duplicitní podstruktury s ekvivalentními přechody. Existují dokonce typy automatů, které nelze těmito standardními technikami minimalizovat vůbec. Tato práce představuje nový přístup k minimalizaci automatů založený na transformaci NKA na nedeterministický zásobníkový automat (NZA). Tato transformace identifikuje skupinu podobných podstruktur a nahradí ji jednou společnou strukturou (procedurou). Tímto způsobem jsme byli schopni zredukovat automaty až o dalších 67.3%. Myšlenka transformace NKA na NZA lze přirovnat k transformaci sekvenčního programu na program, který využívá funkce a zásobníkem volání.

Keywords

Nondeterministic Finite Automata, Nondeterministic Pushdown Automata, Minimization, Network Intrusion Detection Systems, Regular Expressions

Klíčová slova

Nedeterministické Konečné Automaty, Nedeterministické Zásobníkové Automaty, Minimalizace, Systémy Detekce Průniků v Síti, Regulární Výrazy

Reference

ŠEDÝ, Michal. *Repetitive Substructures for Efficient Representation of Automata.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

Nedeterministické konečné automaty (NKA) poprvé představili Michel Reabin a Dana Scot [32]. Jak již jméno automatů napovídá, NKA disponují schopností nedeterministicky přejít do více stavů na základě stejného vstupního symbolu. To umožňuje kompaktnější reprezentaci jazyka. Tato vlastnost však činí operace, jakými jsou minimalizaci, výpočet inkluze nebo výpočet komplementu automatu, obtížnými. Navzdory tomu jsou však nedeterministické konečné automaty široce využívány v mnoha oblastech informatiky, například pro reprezentaci regulárních výrazů, pro monitorování vysokorychlostních sítí [38, 9], v abstraktním regulárním model checkingu [7], k ověřování programů manipulujících s řetězci [2], pro rozhodování procedur logik WS1S a WS2S [16, 20], lineární aritmetiky celých čísel nebo temporálních logik. NKA jsou dokonce používány v bioinformatice k vyhledávání sekvencí nukleotidů v DNA [4].

Automaty s velkým množstvím stavů mohou v řadě algoritmů vést k exponenciálnímu nárůstu stavového prostoru. Zmírnění tohoto problému a snížení výpočetních nároků lze dosahnout jejich minimalizací. Nejznámější minimalizační technikou je slučování stavů [5, 8, 27], které hledá dva jazykově ekvivalentní stavy a ty následně sloučí v jeden. Dalším přístupem je prořezávání hran přechodů [8, 12], které odstraňuje přechody stavů se slabším jazykem v případě, že existuje jazykové silnější stav obsahující ekvivalentní prechod. Opačným přístupem k prořezávání přechodů je přidávání přechodů (saturace) [5, 12], kde nově přidané přechody mohou umožnit další slučování stavů nebo prořezávání přechodů a tím zvýšit efektivitu minimalizace.

Zmíněné metody dovedly, na automatech použitých v našich experimentech, snížit velikost automatů v průměru až na polovinu. Nicméně, ve výsledných automatech se stále mohou nacházet duplicitní podstruktury s podobnými přechody, které tyto metody nedovedou eliminovat. Existují také druhy automatů, které nelze současnými metodami minimalizovat vůbec. Často se jedná o automaty s lineární strukturou (bez větvení) nebo automaty popisující regulární výrazy. Například automaty reprezentující regulární výrazy o dvou částech, kde první část následuje před druhou nebo opačně (například regulární výraz `(.*new XMLHttpRequest.*file://)|(.*file://.*new XMLHttpRequest)`), nebo automaty reprezentující slova s konstantním infixem a stejným prefixem a sufixem (například *abba*, *cbbc*). V těchto případech nemohou slučování stavů, prořezávání hran přechodů ani saturace automaty minimalizovat, protože tyto redukční metody jsou založeny na jazykových inkluzích a uvedené automaty obsahují jen několik, nebo dokonce žádné stavy v jazykové inkluzi.

Tato práce popisuje nový algoritmus minimalizace, který využívá podobnosti sekvence přechodů ve dvou podstrukturách k redukci velikosti automatu. Přístup zahrnuje převod NKA na nedeterministický zásobníkový automat (NZA), přičemž identifikuje podobné podstruktury a ty reprezentuje pouze jednou takzvanou procedurou, která vužívá symbol na zásobníku k určení původní podstruktury automatu, kde se výpočet nachází a kde se má řízení po ukončení procedury vrátit. Cílem úspěšné transformace je nahradit takové sekvence přechodů, aby úspora z jejich redukce převýšila náklady na operace se zásobníkem.

Proces nahrazení více podobných podstruktur jednou společnou podstrukturou (procedurou) začíná výpočtem superproduktu automatu, který identifikuje všechny podobné podstruktury. Superprodukt je vypočítán jako produkt automatu se sebou samým, kde všechny stavy jsou jak počáteční, tak koncové. Následně algoritmus hledá v superproduktu podgraf reprezentující dvě podobné podstruktury. Tyto podgrafy představují jednotlivé kandidáty na vytvoření různých procedůr. Je klíčové vybrat takový podgraf, který povede

k největří redukci velikosti automatu. Zvolený podgraf je následně použit jako šablona pro vytvoření procedury. Algoritmus pokračuje v tomto procesu hledání podgrafů, dokud nejsou všechny podobné podstruktury, které umožňují další redukci automatu, nahrazeny procedurami.

Při použití tohoto nového přístupu jako doplňku standardních minimalizačních technik, implementovaných v nástroji RABIT/Reduce [13], bylo možné dosáhnou následujících dodatešných redukcí v počtu stavů a přechodů. V půměru bylo dosaženo redukce počtu stavů o 29.5% a počtu přechodů o 30.9%. Na skupině automatů, které byly vygenerovány z regulárních výrazů, tvořících pravidla pro systém detekce průniků v síti Snort [37], dosáhla aplikace našeho algoritmu na výsledky nástroje RABIT/Reduce dodatečné redukce až 44.5% v počtu stavů a 60.3% v počtu přechodů. Další významná redukce se ukázala na automatech vygenerovaných v průběhu výpočtu nástroje Z3-noodler [24], kde využití našeho algorithmu na výsledcích nástroje RABIT/Reduce dosáhlo dodatečné redukce až 64.9% v počtu stavů a 67.3% v počtu přechodů.

Transformaci NKA na NZA lze chápat jako převod čistě sekvenčního programu na program s funkcemi a zásobníkem volání. Stejně jako v NZA je v programu použit zásobník k uchování informace o aktuálním stavu, ze kterého byla procedura volána, a stav, kam se má řízení po ukončení procedury vrátit.

Plán práce

Kapitola 2 popisuje modely automatů, jakými jsou NKA a NZA. Dále jsou definovány nové pomocné abecedy, jakými jsou přechodová abeceda a zásobníková abeceda stavu. V kapitole 3 jsou představeny standardní techniky minimalizace používané nástrojem RABIT/Reduce, jakými jsou slučování stavů, prořezávání přechodů a saturace, a rovněž jejich omezení. Kapitola 4 definuje kandidáta procedury, zisk kandidáta procedury a pět typů přechodů, které se přímo pojí s kandidátem procedury. Kapitola také popisuje postup hledání kandidáta procedury s nejvyšším ziskem. Kapitola 5 je vyhrazena detailnímu popisu algoritmu transformace NKA na NZA. Výsledky experimentů na různých typech automatů jsou prezentovány v kapitole 6. Shrnutí práce a možnosti budoucího výzkumu jsou uvedeny v poslední kapitole 7.

Repetitive Substructures for Efficient Representation of Automata

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. 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 16, 2024

Acknowledgements

I would like to thank my supervisor, doc. Lukáš Holík, for his guidance and support throughout the work on this thesis.

Contents

Chapter 1

Introduction

Nondeterministic finite automata (NFAs) were first introduced by Michael Rabin and Dana Scott in [32]. As the name suggests, NFAs have the ability to make nondeterministic transition to multiple states based on the same input symbol. This allows NFAs to represent the language in a more compact way. Despite the fact that this feature makes the minimization and other operations such as inclusion or complementation of NFAs difficult, they find applications in many fields of computer science, such as for the representation of regular expressions, monitoring high-speed networks [38, 9], in abstract regular model checking [7], in verifying programs that manipulate strings [2], or in decision procedures of WS1S and WS2S logic [16, 20], linear integer arithmetic, or temporal logics. NFAs are even used in bioinformatics to search for nucleotide sequences in DNA [4].

Automata with a large number of states can lead, in many algorithms, to an exponential increase in the state space. To address this issue, and reduce computational resources, minimization techniques are used. The most well-known technique is state merging [5, 8, 27], which searches for two language equivalent states and merges them into one. Another approach is transition pruning [8, 12], which removes transitions from weak states if they already exist in some strong state. The opposite of transition pruning is transition addition (saturation) [5, 12], where newly added transitions may allow further state merging or transition pruning.

The mentioned standard minimization methods can reduce the size of most automata used in our benchmarks by up to half. However, the resulting automata can still contain duplicate substructures with similar transitions. There are also types of automata that cannot be minimized by these methods at all. These automata often describe regular expressions or have a purely linear structure (no branching). For example, automata representing regular expressions consisting of two parts where the first part can go before the second part or vice versa (e.g., `(.*new XMLHttpRequest.*file://)|(.*file://.*new XMLHttpRequest)`), or automata representing words with a given infix, the same prefix, and suffix (e.g., *abba*, *cbbc*). State merging, transition pruning, and saturation cannot effectively minimize such automata because these reduction methods are based on language inclusions between states, whereas the mentioned automata contain only a few or even no language inclusions.

In this work, we present a novel minimization algorithm that utilizes the similarity of transitions in duplicate substructures to reduce the size of the automaton. The approach involves the conversion of an NFA to a nondeterministic pushdown automaton (NPDA). It identifies multiple occurrences of similar substructures and replaces them with one common structure (which we call a procedure) that uses a symbol stored on the stack to determine

the original substructure where the calculation is located. The goal of a transformation is to replace such substructures so that the savings from their reduction exceed the overhead of stack operations. Using our minimization approach as a post-processing step of the standard minimization techniques implemented in the RABIT/Reduce tool [13], we achieved the following additional reductions. On average, we achieved a 29.5% reduction in the number of states and a 30.9% reduction in the number of transitions. Additionally, we reduced the size of the automata created from regular expressions that define rules for network intrusion detection system Snort [37] by up to 44.5% in the number of states and 60.3% in the number of transitions. This reduction would significantly decrease the memory requirements for Snort rules and speedup the scanning of high-speed network traffic. We also reduced the size of the automata generated during the calculation of the Z3-noodler tool [24] by up to 64.9% in the number of states and 67.3% in the number of transitions.

The process of replacing multiple similar substructures with one common structure (procedure) begins with calculating a superproduct of an automaton to identify all similar substructures. The superproduct is computed as a product of the automaton with itself, where all states are both initial and final. Subsequently, the algorithm searches within the superproduct for the subgraph representing two similar substructures that are candidates for replacement with a procedure. It is crucial to select a subgraph that has the potential to reduce the size of the automaton the most. The subgraph is then used as a template for the procedure. The algorithm continues to search for similar substructures in the automaton until there are no more substructures that can further reduce the size of the automaton.

The transformation of an NFA to an NPDA can be understood as the conversion of a purely sequential program to a program that uses functions and a call stack. Similarly to the NPDA, a call stack is used in the program to maintain information about the current branch and where the calculation should return to after the procedure ends.

Plan of the Thesis

In Chapter 2, we introduce the basic concepts of nondeterministic finite automata, nondeterministic pushdown automata and new auxiliary alphabets like a transition alphabet or a stack alphabet of a state. Chapter 3 presents the standard minimization techniques, such as state merging, transition pruning, and saturation, that are used within the RABIT/Reduce tool and their limitations. Chapter 4 defines the terms, such as procedure candidate, gain of the procedure candidate, and five types of transitions that are associated with the procedure candidate. It also describes the algorithm for finding the procedure candidate with the highest gain. The core algorithm describing an efficient transformation of NFA to NPDA is presented in Chapter 5. The results of experiments on various types of automata are presented in Chapter 6. Finally, Chapter 7 summarizes the work and outlines the possibilities for future research and utilization of the proposed minimization technique.

Chapter 2

Preliminaries

A spectacular size reduction of an NFA can be obtained by transforming it into a language-equivalent NPDA, representing substructures with similar transition sequences with one procedure. However, before we begin searching for these repeating substructures, it is crucial to have a clear understanding of fundamental terms such as NFA, NPDA, and other auxiliary terms necessary for the transformation concept. It is recommended to go through this chapter to understand the terms used in the rest of the thesis such as a transition alphabet and a stack alphabet of a state.

2.1 Nondeterministic Finite Automaton

Nondeterministic finite automata are a special type of finite state machine that can transition from one state to multiple new states based on the same input (being in several states simultaneously). This nondeterministic property allows NFAs to represent information in a compact form. In contrast, deterministic finite automata (DFAs) are not allowed to make a transition into more than one state, making them less compact than NFAs (an example can be seen in Figure 2.1).

We will consider, for the rest of this section, an NFA as a 5-tuple $M = (Q, \Sigma, \delta, I, F)$.

Definition 2.1. ***Nondeterministic Finite Automaton*** *(NFA) is defined as a 5-tuple $M = (Q, \Sigma, \delta, I, F)$, where its elements are defined as follows:*

- *Q is a finite nonempty set of states,*
- *Σ is a finite alphabet,*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function,*
- *$I \subseteq Q$ is a nonempty set of initial states, and*
- *$F \subseteq Q$ is a set of final states.*

In the context of a transition function δ, we represent a transition from state q to state r over the symbol $a \in \Sigma$ as $r \in \delta(q, a)$. The transition function δ can be generalized to a set of symbols $A \subseteq \Sigma$, a set of states $S \subseteq Q$, or both. Let $q \in Q$. Then, for a set of symbols A, we define $\delta(q, A) = \bigcup_{a \in A} \delta(q, a)$. Similarly, for a set of states S, we define $\delta(S, a) = \bigcup_{s \in S} \delta(s, a)$. Finally, the transition function for a set of symbols and a set of states is defined as $\delta(S, A) = \bigcup_{s \in S} \delta(s, A)$.

It is also important to introduce the *inverse transition function* $\delta^{-1} : Q \times \Sigma \to 2^Q$, where $q \in \delta^{-1}(r, a) \iff r \in \delta(q, a)$. To simplify notation, we will use $succ(q) = \delta(q, \Sigma)$ and $pred(q) = \delta^{-1}(q, \Sigma)$ for the set of *successor* and *predecessor* states of the state $q \in Q$.

Minimal DFA. Minimal NFA.

Figure 2.1: A comparison of the minimal forms of NFA and DFA for the regular language defined by the expression $(a|b|c)^*c$. In the DFA, each occurrence of the symbol c must be treated as a possible end of a word. However, if a or b follow c, it must transition back to the initial state. On the other hand, the NFA can nondeterministically guess that the symbol c is at the end of the word, allowing for a more compact representation.

Definition 2.2. ***Configuration*** *of an NFA is a pair from* $Q \times \Sigma^* =: C$*. Where the pair* (q, w) *indicates that the automaton is in the state* q *and that the unprocessed string* w *remains on the input.*

Definition 2.3. ***Transition*** *is a binary relation on configurations* $\vdash \subseteq C \times C$*, that is defined as* $(q, aw) \vdash (r, w) \iff r \in \delta(q, a)$*, where* $q, r \in Q$*,* $w \in \Sigma^*$*,* $a \in \Sigma$*.*

Let $\chi_0, \chi_1, \ldots, \chi_n$ be a sequence of configurations where $\chi_{i-1} \vdash \chi_i$. Then, an NFA M makes n transitions from χ_0 to χ_n, written as $\chi_0 \vdash^n \chi_n$. If $1 \leq n$, then we write $\chi_0 \vdash^+ \chi_n$. If $0 \leq n$, then we write $\chi_0 \vdash^* \chi_n$.

An automaton can contain *dead states*, whose existence or absence does not affect the language of the automaton. We define two subtypes of dead states: *unreachable states* and *unfinishable states*. The most straightforward reduction in automata size is to remove these dead states, which can be identified using the Tarjan algorithm for the calculation of strongly connected components [40].

Definition 2.4. *Let* $i \in I$ *be an initial state, and* $w \in \Sigma^*$*. The state* $q \in Q$ *is an* ***unreachable state*** *if there is no configuration* (i, w) *such that* $(i, w) \vdash^* (q, \varepsilon)$*.*

Intuitively, an unreachable state is one that cannot be reached from the initial state. This state does not affect the language of the automaton because a language consists of words that start in the initial state and can reach the final state. Clearly, unreachable states will never be reached using such words. A state that is not unreachable is called *reachable*.

Definition 2.5. *Let* $f \in F$ *be the final state and* $w \in \Sigma^*$*. The state* $q \in Q$ *is an* ***unfinishable state*** *if there is no configuration* (q, w) *such that* $(q, w) \vdash^* (f, \varepsilon)$*.*

Intuitively, an unfinishable state is one that cannot reach the final state. Similarly to unreachable states, unfinishable states do not affect the language of the automaton and therefore can be removed. A state that is not unfinishable is called *finishable*.

To apply minimization techniques, such as state merging, transition pruning, or saturation, it is necessary to understand the relations between states. To obtain this information, the state language is utilized. We will define forward and backward languages of states. Moreover, we will also provide definitions of languages of states up to a specified distance.

Definition 2.6. ***Forward language*** *of a state* $q \in Q$ *is a set of words that lead from* q *into* $f \in F$*, formally defined as* $\overrightarrow{L}(q) = \{w \in \Sigma^* \mid (q, w) \vdash^* (f, \varepsilon)$*, where* $f \in F\}$*.*

The forward language of a state can be limited to a specific distance. Let $n \in \mathbb{N}$. Then, the *forward language of a state up to a distance* n is a set of words up to length n that lead from q, formally $\overrightarrow{L}_n(q) = \{w \in \Sigma^* \mid (q, w) \vdash^m (s, \varepsilon),\ \text{where } 0 < m \leq n, s \in Q\}$.

Definition 2.7. ***Backward language*** *of a state $q \in Q$ is a set of words that lead from $i \in I$ into q, formally defined as $\overleftarrow{L}(q) = \{w \in \Sigma^* \mid (i, w) \vdash^* (q, \varepsilon),\ \text{where } i \in I\}$.*

Similarly to the forward language, the backward language of a state can be limited to a specific distance too. Let $n \in \mathbb{N}$. Then, the *backward language of a state up to a distance* n is a set of words defined as $\overleftarrow{L}_n(q) = \{w \in \Sigma^* \mid (s, w) \vdash^m (q, \varepsilon),\ \text{where } 0 < m \leq n, s \in Q\}$.

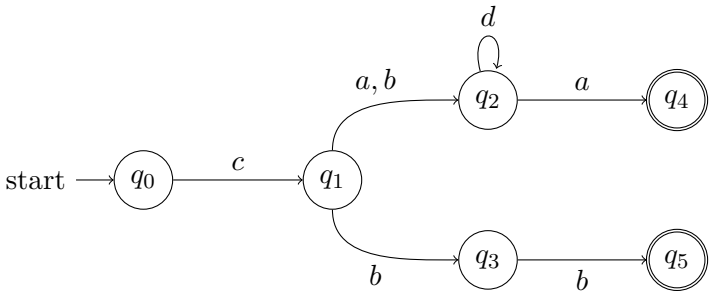


Figure 2.2: An example of state languages. The forward language of state q_1 is defined as $\overrightarrow{L}(q_1) = \{ad^*a, bd^*a, bb\}$, and the forward language limited to a distance of 1 is $\overrightarrow{L}_1(q_1) = \{a, b\}$. The backward language of state q_4 is $\overleftarrow{L}(q_4) = \{cad^*a, cbd^*a\}$, and the backward language limited to a distance of 2 is $\overleftarrow{L}_2(q_4) = \{a, aa, ba, da\}$.

Definition 2.8. ***Language of an automaton*** *M is the set of words that lead from an initial state to a final state, formally defined as $L(M) = \bigcup_{i \in I} \overrightarrow{L}(i)$.*

Two NFAs M_1 and M_2 have *equal language* if $L(M_1) \equiv L(M_2)$. That is, for each word $w_1 \in L(M_1)$, there exists a sequence of transitions $(i_2, w_1) \vdash^* (\varepsilon, f_2)$, where $i_2 \in I_2$ and $f_2 \in F_2$, and for each word $w_2 \in L(M_2)$, there exists a sequence of transitions $(i_1, w_2) \vdash^* (\varepsilon, f_1)$, where $i_1 \in I_1$ and $f_1 \in F_1$.

2.2 Simulation

To perform a reduction of an NFA by state merging, transition pruning, or saturation, it is necessary to determine the relation between the languages of states. However, it is often too expensive to precisely calculate the language inclusion. Therefore, the approximation method called simulation [25, 3] is used. If the simulation says that a state r is simulated by state q ($r \overrightarrow{\preceq} q$), then the forward language of state r is included in the forward language of state q, but not vice versa. That is, there are language inclusions that the simulation does not detect, as shown in Figure 2.3.

Definition 2.9. ***Simulation*** *on an NFA M is a preorder $\preceq \subseteq Q \times Q$. Let $a \in \Sigma$. The relation $r \preceq q$ only exists if $r \in F \implies q \in F$ and for every $a \in \Sigma$ such that $r' \in \delta(r, a)$, there exists $q' \in \delta(q, a)$, which must further satisfy $r' \preceq q'$.*

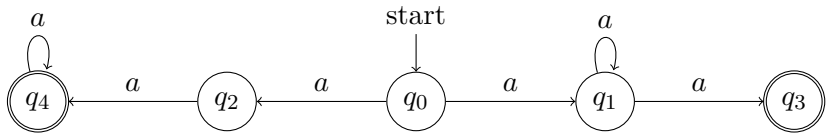


Figure 2.3: The simulation does not detect the language inclusion $\overrightarrow{L}(q_2) \subseteq \overrightarrow{L}(q_1)$, although their languages are equivalent. This is caused by the self-loop on the final state q_4, which prevents the state q_3 from simulating the state q_4.

2.3 Nondeterministic Pushdown Automaton

In order to describe languages more complex than regular languages $\mathcal{L}_3$, the nondeterministic pushdown automaton (NPDA) has been introduced. Although the concept of a pushdown tape (stack) has been used since 1954 [23], NPDAs were first formalized between 1962 and 1963 by Chomsky [10] and Evey [17]. A nondeterministic pushdown automaton (NPDA) is a generalization of a nondeterministic finite automaton (NFA) with a stack where additional information can be stored. Consequently, the transition between the automaton's states is performed not only on the basis of the input but also on the value of an element on the top of the stack. This enables NPDA to recognize context-free languages $\mathcal{L}_2$. For example, the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$, which contains words over an alphabet $\{a, b\}$ where the first half of the word consists of n symbols a and the second half consists of n symbols b. In our work, we utilize a stack to store the information about the current branch of the automaton, allowing us to determine the point where to return at the end of the procedure. This concept is identical to a call stack in programming languages.

There exist many variants of NPDAs. Similarly to NFAs, there is a deterministic variant of NPDAs. However, DPDAs are weaker than their nondeterministic counterparts. For example, DPDAs cannot represent palindromes of even length. NPDAs can also differ in their acceptance conditions. They can accept a word by emptying the stack, reaching the final state, or reaching the final state with a specific symbol on the top of the stack. In our work, we employ the last acceptance condition, where a specific symbol is required to be on the top of the stack for acceptance.

In this section, we will view an NPDA as an 8-tuple $M = (Q, \Sigma_\varepsilon, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$.

Definition 2.10. ***Nondeterministic Pushdown Automaton*** *(NPDA) is defined as an 8-tuple $M = (Q, \Sigma_\varepsilon, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, with its elements defined as follows:*

- *Q is a finite nonempty set of states,*
- *Σ_ε is a finite alphabet containing the symbol ε (the empty string),*
- *Γ_ε is a finite stack alphabet containing the symbol ε,*
- *$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to 2^{Q \times \Gamma_\varepsilon}$ is a transition function,*
- *$I \subseteq Q$ is a nonempty set of initial states,*
- *$\lambda : I \to 2^{\Gamma_\varepsilon}$ is a function of initial stack symbols,*
- *$F \subseteq Q$ is a set of final states, and*
- *$\phi : F \to 2^{\Gamma_\varepsilon}$ is a function of final stack symbols.*

Because the nondeterministic variant of a pushdown automaton can start in an arbitrary initial state, we have to specify the set of initial stack symbols from which the automaton

can nondeterministically choose for each initial state. If $\lambda(i) = \{\varepsilon\}$, then the automaton starts in the state i with an empty stack. On the other hand, if $\lambda(i) = G \neq \{\varepsilon\}$, then the automaton can start in the state i with a stack symbol nondeterministically chosen from G.

Also, since the automaton accepts the word not only by reaching the final state but also by containing a specific symbol at the top of the stack, we have to describe this property by a function ϕ. Let $f \in F$. If $\phi(f) = \{\varepsilon\}$, then the automaton accepts the word by reaching the final state f with an empty stack. In contrast, if $\phi(f) = G \neq \{\varepsilon\}$, then the automaton accepts the word by reaching the final state f while containing the symbol $g \in G$ at the top of the stack.

If there is a transition from state $q \in Q$ to state $r \in Q$ with input symbol $a \in \Sigma_\varepsilon$, and a symbol $\alpha \in \Gamma_\varepsilon$ is popped from the top of the stack while a symbol $\beta \in \Gamma_\varepsilon$ is pushed onto the stack, then we write $(r, \beta) \in \delta(q, a, \alpha)$. Similarly to an NFA, the transition function can also be generalized to a set of symbols $A \subseteq \Sigma_\varepsilon$, $B \subseteq \Gamma_\varepsilon$, a set of states $S \subseteq Q$, or any combination of these sets. For a set of states S, we define $\delta(S, a, \alpha) = \bigcup_{s \in S} \delta(s, a, \alpha)$. Similarly, for other combinations.

The *inverse transition function* δ^{-1} of an NPDA is defined as $(q, \alpha) \in \delta^{-1}(r, a, \beta)$ if and only if $(r, \beta) \in \delta(q, a, \alpha)$, where $q, r \in Q$, $a \in \Sigma_\varepsilon$, and $\alpha, \beta \in \Gamma_\varepsilon$. Note that in the inverse transition function, the push and the pop stack symbols are swapped.

To describe a set of *successors* or *predecessors* of a state $q \in Q$, we use a notation similar to that of an NFA: $succ(q) = \delta(q, \Sigma_\varepsilon, \Gamma_\varepsilon)$ and $pred(q) = \delta^{-1}(q, \Sigma_\varepsilon, \Gamma_\varepsilon)$. Here, Σ_ε represents the input alphabet, and Γ_ε represents the stack alphabet.

Definition 2.11. ***Configuration*** *of an NPDA is a triple from* $Q \times \Sigma^* \times \Gamma^* =: C$*, where the triple* (q, w, β) *means that the machine is in the state* q*, the unprocessed word* w *remains on the input, and the stack contains the word* β *(with the top of the stack on the left).*

Definition 2.12. ***Transition*** *is a binary relation on the set of automaton's configurations* $\vdash \subseteq C \times C$*, defined as* $(q, aw, X\alpha) \vdash (r, w, Y\alpha) \iff (r, Y) \in \delta(q, a, X)$*, where* $q, r \in Q$*,* $w \in \Sigma^*$*,* $\alpha \in \Gamma^*$*,* $a \in \Sigma_\varepsilon$*, and* $X, Y \in \Gamma_\varepsilon$*, with* X *being the symbol popped from the stack and* Y *being the symbol pushed onto the stack.*

The complex transition definition describes a transition from the automaton's configuration $(q, aw, X\alpha)$ to $(r, w, Y\alpha)$, meaning that by applying the transition rule specified by the transition function δ, the automaton can move from state q to state r, by reading the symbol a from the input (leaving the rest of the input w unprocessed) and changing the top of the stack (from X to Y).

Definition 2.13. ***Forward language*** *of a state* $q \in Q$ *is a set of words over the alphabet* Σ *defined as* $\overrightarrow{L}(q) = \{w \in \Sigma^* \mid \exists \alpha \in \Gamma^* : (q, w, \alpha) \vdash (f, \varepsilon, \beta) \wedge \beta \in \phi(f),$ where $\beta \in \Gamma_\varepsilon, f \in F\}$.

The forward language of a state q contains all words w that start in the configuration (q, w, α) with an arbitrary stack $\alpha \in \Gamma^*$, so that they reach the final configuration (f, ε, β) in the final state $f \in F$ and meet the condition $\beta \in \phi(f)$. Similarly to NFA, we can define the forward language $\overrightarrow{L}_n(q)$ of a state q limited by a maximal distance $n \in \mathbb{N}$.

Definition 2.14. ***Backward language*** *of a state* $q \in Q$ *is a set of words over* Σ *defined as* $\overleftarrow{L}(q) = \{w \in \Sigma^* \mid \exists i \in I : \exists \alpha \in \lambda(i) : (i, w, \alpha) \vdash (q, \varepsilon, \beta),$ where $i \in I, \beta \in \Gamma^*\}$.

The backward language of a state q contains all words w that start in the initial state $i \in I$ with an initial stack symbol $\alpha \in \lambda(i)$ and can reach the configuration (q, ε, β) in the state q with an arbitrary stack content β. Furthermore, we can also define the backward language $\overleftarrow{L}_n(q)$ of a state q limited by a maximum distance $n \in \mathbb{N}$.

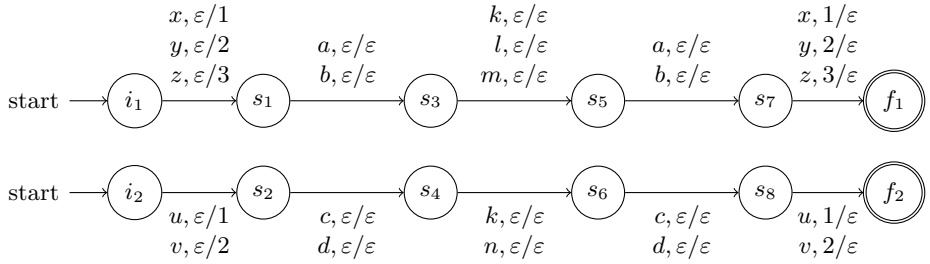
Definition 2.15. ***Language of the automaton*** *M is the set of words accepted by the automaton, defined as $L(M) = \bigcup_{i \in I} \overrightarrow{L}(i)$, where I is the set of initial states.*

To evaluate the reduction effect of representing repetitive substructures with a single procedure, we have to determine how many symbols of Σ are used within a transition. To do so, we define the transition alphabet.

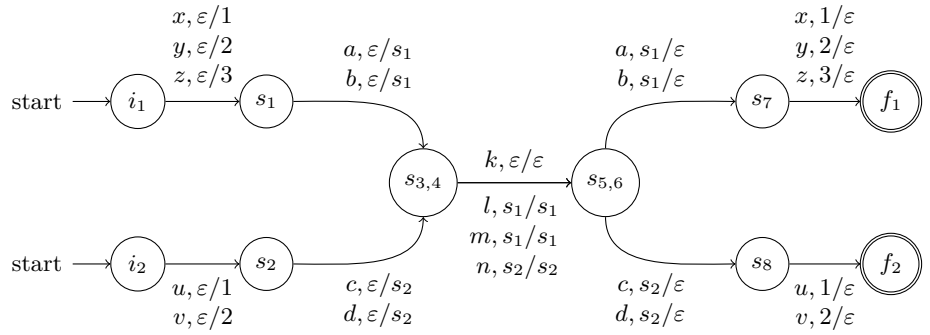
Definition 2.16. ***Transition alphabet*** *of a transition between states q and r from Q is a set of symbols from Σ that label the transition, defined as $\sigma(q, r) = \{a \in \Sigma \mid \exists \alpha, \beta \in \Gamma_\varepsilon : (r, \beta) \in \delta(q, a, \alpha)\}$.*

The transition alphabet $\sigma(q, r)$ contains all symbols that are used within the transition between states q and r. However, it is also useful be able to determine a set of symbols that label the transition between states q and r without accessing the stack (the pop and push symbols are equal to ε). This set is called the *epsilon transition alphabet* and is defined as $\sigma_{\varepsilon/\varepsilon}(q, r) = \{a \in \Sigma \mid (r, \varepsilon) \in \delta(q, a, \varepsilon)\}$. Another transition alphabet is called the *co-epsilon transition alphabet*, denoted as $\overline{\sigma_{\varepsilon/\varepsilon}}$. It is a complement of $\sigma_{\varepsilon/\varepsilon}$ containing symbols used on transitions that involve stack operations (push or pop).

Definition 2.17. ***Stack alphabet of a state*** *$q \in Q$ is a set of stack symbols that can appear on the top of the stack while the automaton is in the state q. It is formally defined as $\gamma(q) = \{\alpha \in \Gamma_\varepsilon \mid \exists w \in \Sigma^* \, \exists i \in I : \exists \beta \in \lambda(i) : (i, w, \beta) \vdash^* (q, w', \alpha\eta)$, where $w' \in \Sigma^*, \eta \in \Gamma^*\}$.*



NPDA with two procedures that has similar transitions.



NPDA with new procedure within procedures.

Figure 2.4: The creation of a procedure within procedures using another stack level significantly increases the number of stack operations and therefore is omitted in our work.

As we have already mentioned, NPDAs model context-free languages $\mathcal{L}_2$. However, $\mathcal{L}_2$ are not closed under intersection and complement operations. Let $A = \{a^n b^n c^m \mid m, n \in \mathbb{N}_0\}$

and $B = \{a^m b^n c^n \mid m, n \in \mathbb{N}_0\}$ be two context-free languages. The intersection of these languages $A \cap B$ results in a non-context-free language $C = \{a^n b^n c^n \mid n \in \mathbb{N}_0\}$. To prove that $C \notin \mathcal{L}_2$, we can use the pumping lemma for context-free languages, also known as the Bar-Hillel lemma [29]. As a result, $\mathcal{L}_2$ cannot be closed under the complementation as well because the intersection could then be expressed using the De Morgan laws as $A \cap B = \overline{\overline{A} \cup \overline{B}}$. Another problem of context-free languages A and B is the undecidability of language inclusion $A \subseteq B$, universality $A = \Sigma^*$, or disjointness of two languages $A \cap B = \emptyset$, which has been shown in [26].

Due to the undecidable nature of some properties of $\mathcal{L}_2$, we decided to use a weaker variant of NPDA, where the stack is limited to contain a maximum of $n \in \mathbb{N}$ symbols for every procedure. This corresponds to the limit on the size of the call stack. We particularly chose n to be equal to 1 (NPDA_1), restricting the stack to contain only one symbol at a time. We tested higher stack limits, allowing us to have multiple immersed procedures, but this approach led to a smaller reduction of the automaton, caused by the increase in the number of transitions manipulating with the stack, as shown in Figure 2.4.

Theorem 2.1. *The language of a nondeterministic pushdown automaton with a stack limited to maximum of one symbol ($NPDA_1$) is a regular language.*

Proof. We will divide this proof into two parts. First, we will show that the language $L(NPDA_1) \subseteq \mathcal{L}_3$. Then, we will show that $\mathcal{L}_3 \subseteq L(NPDA_1)$.

1. $L(NPDA_1) \subseteq \mathcal{L}_3$:
   Fore every NPDA_1 $M_P = (Q_P, \Sigma_\varepsilon, \Gamma_\varepsilon, \delta_P, I_P, \lambda, F_P, \phi)$, there exists its language equivalent NFA $M_F = (Q_F, \Sigma_\varepsilon, \delta_F, I_F, F_F)$, where:
   - $Q_F = Q_P \times \Gamma_\varepsilon$,
   - $\delta_F((q, \alpha), a) = \{(r, \beta) \in Q_F \mid (r, \beta) \in \delta_P(q, a, \alpha)\} \cup \{(r, \alpha) \in Q_F \mid (r, \varepsilon) \in \delta_P(q, a, \varepsilon)\}$,
   - $I_F = \{(i, \alpha) \in Q_F \mid i \in I_P \wedge \alpha \in \lambda(i)\}$, and
   - $F_F = \{(f, \alpha) \in Q_F \mid f \in F_P \wedge \alpha \in \phi(f)\}$.

2. $\mathcal{L}_3 \subseteq L(NPDA_1)$:
   Fore every NFA $M_F = (Q, \Sigma_\varepsilon, \delta_F, I, F)$ representing regular language, there exists its language equivalent NPDA_1 $M_P = (Q, \Sigma_\varepsilon, \Gamma_\varepsilon, \delta_P, I, \lambda, F, \phi)$, where:
   - $\Gamma_\varepsilon = \{\varepsilon\}$,
   - $\delta_P(q, a, \varepsilon) = \delta_F(q, a) \times \{\varepsilon\}$ for $q \in Q, a \in \Sigma$,
   - $\lambda(i) = \{\varepsilon\}$ for $i \in I$, and
   - $\phi(f) = \{\varepsilon\}$ for $f \in F$.

□

As mentioned before, the DPDA is significantly weaker than the NPDA. However, when using a stack limited to a maximum of one symbol, NPDA_1 is equivalent to DPDA_1.

Theorem 2.2. *For each $NPDA_1$ M, there exists a $DPDA_1$ M' such that $L(M) \equiv L(M')$.*

Proof. Since the languages accepted by NPDA_1 and NFA are equivalent, as stated by Theorem 2.1, every NPDA_1 can be represented as an NFA. Subsequently, this NFA can be determinized and transformed back into a DPDA_1. Furthermore, because every DPDA_1 is in fact NPDA_1, it follows that $L(NPDA_1) \equiv L(DPDA_1)$. □

By modifying the proofs of Theorems 2.1 and 2.2, it is also possible to demonstrate that NPDA_n describes regular languages and is equivalent to DPDA_n for any fixed $n \in \mathbb{N}_0$.

Chapter 3

Standard Minimization Techniques

The concept of minimizing the size of automata is as old as automata theory itself. In this chapter, we will describe the most commonly used minimization techniques and their limitations. The oldest and most well-known minimization approach is state merging [5, 8, 27], which utilizes language inclusion to detect states whose language is already represented by some other state, thus allowing these two states to merge together. The second method is transition pruning [8, 12], which identifies, using language inclusion, redundant transitions that can be removed from the automaton without changing its language. Last but not least, saturation [12] is a method that, in contrast to transition pruning, adds transitions to the automaton without changing its language, revealing new language inclusions and allowing further application of state merging or transition pruning.

Although standard minimization techniques for NFAs, based on language inclusions, can efficiently reduce automata, they are not omnipotent and can leave redundant substructures in the automata or may not be capable of reducing the automata at all.

This chapter, inspired by [42], presents minimization theorems for state merging, transition pruning, saturation, and shows how these minimization methods are used to reduce the size of the automaton. It also illustrates in three examples the limitations of these methods and demonstrates why our approach, which uses NPDA_1 instead of NFA, achieves better automata reduction.

To describe minimization methods, a nondeterministic automaton $M = (Q, \Sigma, \delta, I, F)$ will be used throughout this chapter.

3.1 State Merging

The oldest minimization technique, state merging, relies on the inclusion relation among the languages of states, which can be approximated using the simulation. If two states have equivalent backward or forward languages, they can be merged. Even states that are not equivalent but have their backward and forward languages included in the languages of a stronger state, can be merged with that stronger state. When two states q and r are merged, a new state m is created, and all transitions associated with states q or r are redirected to the new state m.

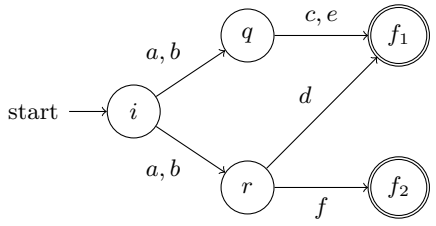
To simplify the notation for redirecting the transitions of states q and r into m, we will use substitution. The substitution of any state q by a state m in A replaces all the occurrences of q in A with m, denoted as $A[m/q]$.

Theorem 3.1. *Two states q and r can be merged into one if at least one of the following conditions is met:*

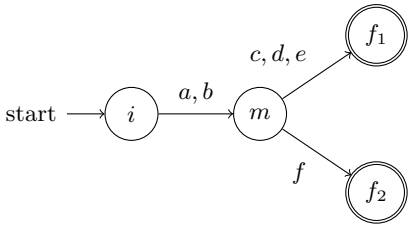
1. $\overleftarrow{L}(q) \equiv \overleftarrow{L}(r)$,
2. $\overrightarrow{L}(q) \equiv \overrightarrow{L}(r)$, *or*
3. $\overleftarrow{L}(q) \subseteq \overleftarrow{L}(r) \wedge \overrightarrow{L}(q) \subseteq \overrightarrow{L}(r)$.

When two states $q, r \in Q$ are merged, they are removed from the automaton and a new state $m \notin Q$, which has not been used before, is created. Transitions involving q and r are substituted with transitions involving m. The new state m inherits the final state status if either q or r was a final state, and it inherits the initial state status if either q or r was an initial state. Formally, the new automaton $M' = (Q', \Sigma, \delta', I', F')$ is created, and its elements are defined as follows:

- $Q' = Q \setminus \{q, r\} \cup \{m\}$,
- $\delta'(s, a) = \begin{cases} \delta(q, a)[m/q, m/r] \cup \delta(r, a)[m/q, m/r] & \text{if } s = m, \\ \delta(s, a)[m/q, m/r] & \text{otherwise} \end{cases}$
- $I' = I[m/q, m/r]$,
- $F' = F[m/q, m/r]$



Automaton with two states q and r that have equivalent backward language.



Resulting automaton with states q and r merged into the state m.

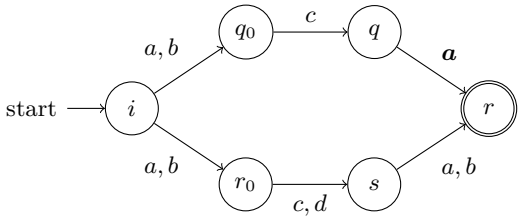
Figure 3.1: An example of state merging utilizing backward language equivalence.

3.2 Transition Pruning

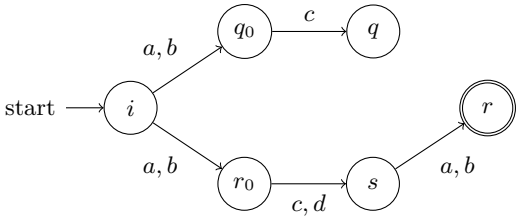
The key idea of transition pruning is to remove transitions that are not necessary in the automaton and thus can be eliminated without altering the language. Unlike state merging, which reduces automaton size by replacing two states with one, transition pruning reduces automaton size by removing transitions and potentially creating unreachable or unfinishable states, which can then be easily eliminated.

Theorem 3.2. *The transition $q \xrightarrow{a} r$ can be removed from the automaton if any of the following conditions are met:*

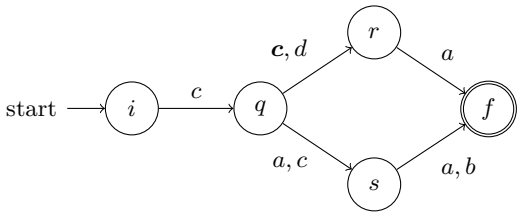
1. $\exists s \xrightarrow{a} r \wedge \overleftarrow{L}(q) \subseteq \overleftarrow{L}(s)$,
2. $\exists q \xrightarrow{a} s \wedge \overrightarrow{L}(r) \subseteq \overrightarrow{L}(s)$, *or*
3. $\exists q' \xrightarrow{a} r' \wedge \overleftarrow{L}(q) \subseteq \overleftarrow{L}(q') \wedge \overrightarrow{L}(r) \subseteq \overrightarrow{L}(r')$.



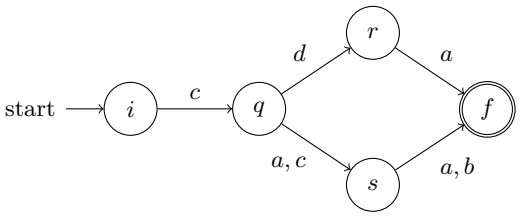
Automaton $M_{3.2.1}$ with the transition $q \xrightarrow{a} r$ which is going to be pruned based on the first condition of Theorem 3.2.



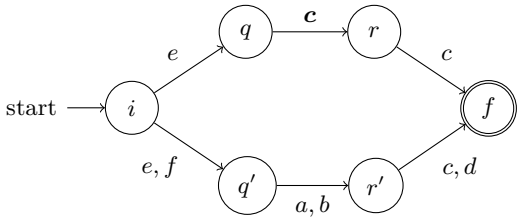
Resulting automaton after pruning the transition $q \xrightarrow{a} r$ in the automaton $M_{3.2.1}$.



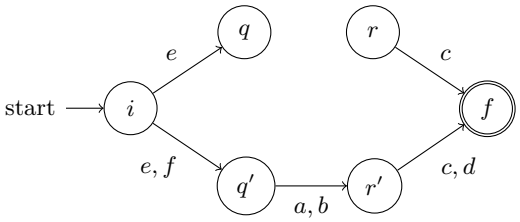
Automaton $M_{3.2.2}$ with the transition $q \xrightarrow{c} r$ which is going to be pruned based on the second condition of Theorem 3.2.



Resulting automaton after pruning the transition $q \xrightarrow{c} r$ in the automaton $M_{3.2.2}$.



Automaton $M_{3.2.3}$ with the transition $q \xrightarrow{c} r$ which is going to be pruned based on the third condition of Theorem 3.2.



Resulting automaton after pruning the transition $q \xrightarrow{c} r$ in the automaton $M_{3.2.3}$.

Figure 3.2: This figure illustrates the transition pruning based on each condition of Pruning Theorem 3.2. It can be observed that besides reducing the number of transitions, transition pruning also aims to create unreachable or unfinishable states that can later be easily removed from the automaton.

3.3 Saturation

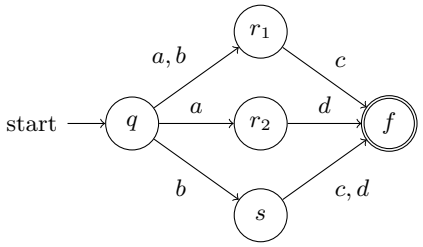
The saturation is opposite to transition pruning. Whereas transition pruning removes transitions, saturation adds more transitions into the automaton (saturates the automaton with transitions) without changing the automaton's language. The newly added transitions may reveal new language inclusions. Saturation is typically used as an extension of the standard minimization flow, which uses state merging in combination with transition pruning. After some time, there are no more states in language inclusion in the automaton. Therefore, no further minimization using merging or pruning can be done. However, by applying saturation, new language inclusions can be created, and further minimization can obtain an automaton with fewer states. On the other hand, saturation also adds transitions that do

not contribute to new language inclusions, and therefore, the resulting automaton can have more transitions than necessary.

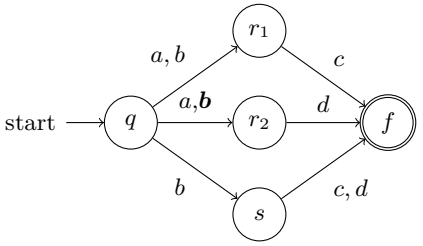
The basic idea of saturation is to add a transition from a state with a stronger language between transitions of a weaker state. This will not change the language of the automaton, because every word that can be in the weaker state and traverse the newly added transition can already go through the original transition connected with the stronger state.

Theorem 3.3. *The transition $q \xrightarrow{a} r$ can be added to the automaton if one of the following conditions is met:*

1. $\exists s \xrightarrow{a} r \wedge \overleftarrow{L}(q) \subseteq \overleftarrow{L}(s)$, *or*
2. $\exists q \xrightarrow{a} s \wedge \overrightarrow{L}(r) \subseteq \overrightarrow{L}(s)$.



So far, the nonminimizable automaton $M_{3.3}$ exhibits only one-sided language inclusion (no equivalence and no bilateral inclusion).



Automaton $M_{3.3}$ after saturation by the transition $q \xrightarrow{b} r_2$ based on the second condition of Theorem 3.3.

Figure 3.3: This figure shows the power of saturation. Without saturation, the automaton $M_{3.3}$ could not be further minimized. However, the addition of the transition $q \xrightarrow{b} r_2$ created new backward language equivalence between states r_1 and r_2, allowing them to merge into the state $r_{1,2}$. Consequently, the states $r_{1,2}$ and s can be merged together, resulting in the automaton with only three states instead of five.

3.4 The Limitation

As state merging, transition pruning, and saturation rely on a language inclusion relation, they can only be applied to automata where such a language relation on states exists. However, there are types of automata that contain many similar or even identical sequences of transitions, which are not detected by language inclusion because of differences in their prefix or suffix. Therefore, minimization based on language inclusion cannot be applied to such automata. On the other hand, these automata represent the main use case for our technique. They often have a pure linear structure, represent regular expressions with repeating substrings, a union of many regular expressions describing similar patterns, or regular expressions with counting[1] (for example, $ab\{100\}c$).

In order to minimize these cases, our approach identifies the repeating subgraphs in the automaton and transforms the NFA into NPDA_1 in such a way that each duplicate subgraph is represented by a corresponding procedure. Because the procedure is a place where the flow from more automaton subgraphs intersect, we have to distinguish which

[1] In some cases, it might be more suitable to represent regular expressions with counting using register automata with counters.

subgraph is currently being mimicked by a procedure and where to return. To preserve this information, the stack is used. It can be seen in the following examples that the utilization of procedures allows us to significantly reduce the size of the automaton.

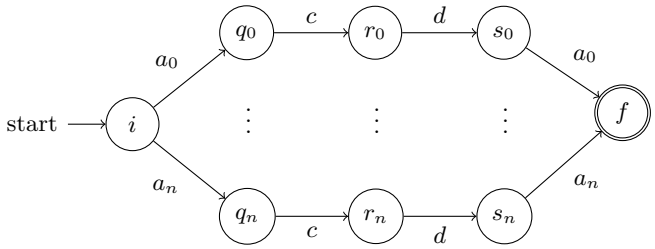


Figure 3.4: An NFA $M_{3.4}$ with a language $\{a_k cda_k \mid 0 \leq k \leq n \in \mathbb{N}_0, \text{ where } a_k, c, d \in \Sigma\}$.

The example of the automaton created as a union of words that contain the same pattern, in our case the infix cd, is presented in Figure 3.4. Each of the n words represented by this automaton contains the same infix cd, but each word has its specific symbol that encloses the infix. The identical infix exists n times in the automaton, making it extremely redundant. Moreover, because of the differences between the prefixes and suffixes of each word, there is no language inclusion that could be used to minimize the automaton using standard methods. However, a procedure can represent the infix cd only once and therefore reduce the automaton by $(n-1) \cdot (|cd|+1)$ states and $(n-1) \cdot |cd|$ transitions.

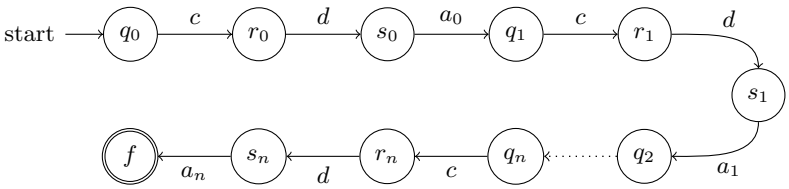


Figure 3.5: An NFA $M_{3.5}$ representing a string $cda_0cda_1 \cdots cda_{n-1}cda_n$ for $n \in \mathbb{N}$.

Another example of an automaton to which standard minimization techniques cannot be applied is presented in Figure 3.5. The automaton $M_{3.5}$ describes a regular language that contains a repeating sequence of the word cd interleaved with a symbol a_k for $k \leq n \in \mathbb{N}_0$. Due to the linear structure of the automaton, it lacks any language inclusion and therefore cannot be minimized using traditional methods. One possible approach to reduce the size of this automaton is to represent n repeating substructures for cd only once using a procedure. The procedure will utilize n stack symbols to determine which substructure of cd it currently represents. By doing so, the automaton can be reduced by the same number of states and transitions as in the previous example.

The last example of an automaton with repeating substructures that cannot be eliminated using standard minimization methods is taken from a real-world application. The automaton in Figure 3.6 originates from one of the rules of the network intrusion detection system Snort [37]. The automaton represents a regular expression `(.*new XMLHttpRequest .*file://)|(.*file://.*new XMLHttpRequest)`, which matches the usage of the XMLHttpRequest object in combination with the file protocol. The automaton consists of two branches that are identical except for the order of the substrings `.*new XMLHttpRequest` and `.*file://`. These two different orders cause the automaton to lack any language inclusion (except for `.*`) that could be used to merge these two branches.

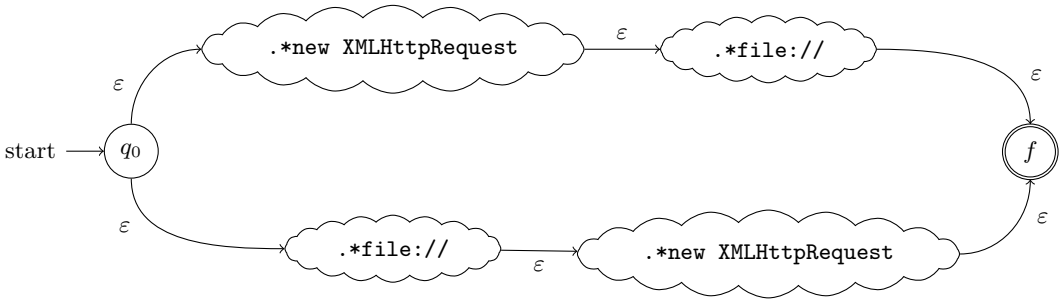


Figure 3.6: An NFA $M_{3.6}$ with duplicate substructures representing the regular expression `(.*new XMLHttpRequest.*file://)|(.*file://.*new XMLHttpRequest)`.

The procedures can represent the repeating substructures for `.*new XMLHttpRequest` and `.*file://`, in the automaton $M_{3.6}$, only once and use the stack to determine the order of the substrings currently being used. Using this approach, the automaton can eliminate all redundant substructures. The resulting NPDA_1 is shown in Figure 3.7. Each cloud represents multiple states and transitions modeling a regular expression.

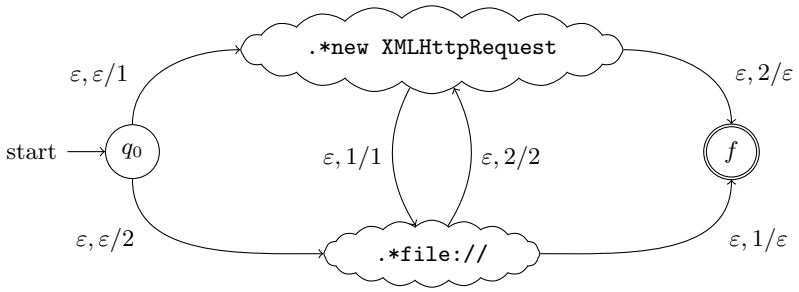


Figure 3.7: The NPDA_1 created from the automaton $M_{3.6}$ by representing repeating substructures for `.*new XMLHttpRequest` and `.*file://` using procedures. The first order of the substrings, starting with `.*new XMLHttpRequest`, is indicated by the stack symbol 1, and the second order, starting with `.*file://`, by the stack symbol 2.

Chapter 4

Procedure Finding

In order to replace multiple occurrences of similar substructures in the automaton by one common structure (which we call a procedure), it is necessary to identify pairs of such substructures that will maximize the reduction of the automaton's size. These pairs are called procedure candidates and will be used as templates for the construction of procedures. The process of selecting a procedure candidate begins with the identification of all similar substructures within the automaton. This involves calculating the superproduct of the automaton, which is a product of the automaton with itself where all states are both initial and final. Subsequently, the procedure candidate, which is a linear subgraph of the superproduct with possibly the highest reduction potential, is selected. Based on this selected procedure candidate, we create the procedure, which is a substructure within the resulting automaton.

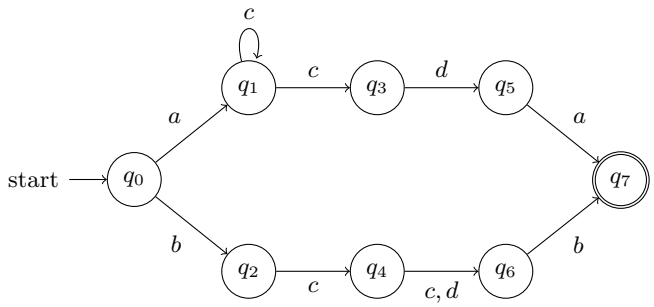
This chapter presents four cornerstones of effective procedure creation. The first defines a superproduct of the automaton that represents all pairs of similar substructures. Subsequently, we present a metric for measuring the reduction potential of such substructures. The third section is dedicated to selecting the linear subgraph of the superproduct with maximal reduction potential. The fourth section defines the procedure candidate and five types of associated transitions. Finally, the last section shows why it is important not to focus solely on the number of automaton states when evaluating the effectiveness of the reduction, but also on the number of stack symbols and transitions.

4.1 Identifying Similar Substructures

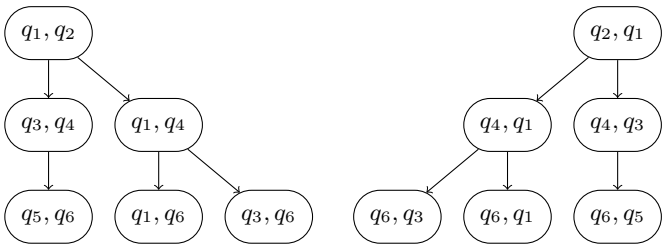
It is necessary to find all similar substructures (transition sequences) to be able to correctly select a procedure with good reduction potential. These substructures cannot be found by looking for graph similarities because the same words can be represented by different structures within an automaton. The similarity of two transition sequences, starting in two different states, is calculated as the intersection of their languages. To do so, we calculate the superproduct of an automaton, which is an unary operation that generalizes an automaton product with itself, where all its states are initial and final.

Definition 4.1. *Let $M = (Q, \Sigma_\varepsilon, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$ be an NPDA$_1$. The **superproduct** of the automaton is an oriented graph $SP(M) = (V, E)$, where:*

- $V = \{(q, r) \in Q \times Q \mid q \neq r\}$, *and*
- $E = \{((u_1, v_1), (u_2, v_2)) \in V \times V \mid |\{u_1, v_1, u_2, v_2\}| = 4 \wedge \sigma_{\varepsilon/\varepsilon}(u_1, u_2) \cap \sigma_{\varepsilon/\varepsilon}(v_1, v_2) \neq \emptyset\}$.



An NFA $M_{4.1}$ with a similar transition sequences within states q_1, q_3, q_5 and q_2, q_4, q_6.



The visualization of the superproduct $\mathrm{SP}(M_{4.1})$, representing similar transition sequences in $M_{4.1}$, without singleton nodes (nodes without edges) for better readability.

Figure 4.1: This figure illustrates an NPDA_1 and its superproduct. It can be seen that the superproduct contains for each node (q_i, q_j) a mirrored node (q_j, q_i). From the superproduct, we can see that the automaton $M_{4.1}$ contains similar transition sequences going over states q_1, q_3, q_5 and q_2, q_4, q_6 for the word cd, over states q_1, q_1, q_1 and q_2, q_4, q_6 for the word cc, and over states q_1, q_1, q_3 and q_2, q_4, q_6 also for the word cc.

To find the best subgraph of a superproduct that contains the procedure candidate promising the maximal reduction, we need to be able to focus on arbitrary subgraph of the superproduct with respect to some node (root). To achieve this, we define rooted subgraph of a superproduct which is a subtree of the superproduct graph.

Definition 4.2. *The **rooted subgraph of a superproduct** $SP(M) = (V, E)$ with a root $r \in V$ is a subgraph $SP_{\hat{r}}(M) = (V_{\hat{r}} \subseteq V, E_{\hat{r}} \subseteq E)$, where the following conditions hold:*

1. $\nexists u \in V_{\hat{r}} : (u, r) \in E_{\hat{r}}$
2. $\forall (u, v) \in E_{\hat{r}} : u \neq r \implies \exists v_0 \in V_{\hat{r}} : (r, v_0) \in E_{\hat{r}} \wedge$
   $\exists n \in \mathbb{N}_0 : (\forall 1 \leq i \leq n : (v_{i-1}, v_i) \in E_{\hat{r}}) \wedge$
   $(v_n, v) \in E_{\hat{r}}$
3. $\forall u \in V_{\hat{r}} : \exists v \in V_{\hat{r}} : (u, v) \in E_{\hat{r}} \vee (v, u) \in E_{\hat{r}}$

Intuitively, the first condition ensures that there are no edges entering the root. According to the second condition, each edge must be reachable from the root. The last condition forces the subgraph to contain only vertices with an incoming or outgoing edges.

Furthermore, the subgraph of a superproduct can be restricted by a maximal distance $d \in \mathbb{N}$ from the root $r \in V$, denoted as $SP^d_{\hat{r}}(M)$. In this case, the existential quantifier $\exists n \in \mathbb{N}_0$ in the second condition is replaced by $\exists 0 \leq n \leq d - 1$.

4.2 The Best Subgraph

A rooted subgraph of a superproduct with the highest gains is likely to achieve the greatest reduction in the automaton. To quantify the reduction potential of a subgraph, we define the gain of an edge and a vertex, indicating the decrease in the number of automaton transitions if the edge or vertex is used in the creation of a procedure.

Definition 4.3. *Let $P = (V, E)$ be a subgraph of a superproduct.* ***Edge gain*** *is a function $G_e : E \to \mathbb{Z}$ defined as follows:*

$$\begin{aligned} G_e((u_1, u_2), (v_1, v_2)) = &|\sigma_{\varepsilon/\varepsilon}(u_1, v_1) \cap \sigma_{\varepsilon/\varepsilon}(u_2, v_2)| - \\ &|\sigma_{\varepsilon/\varepsilon}(u_1, v_1) \setminus \sigma_{\varepsilon/\varepsilon}(u_2, v_2)| \cdot max(0, |\gamma(u_1) - 1|) - \\ &|\sigma_{\varepsilon/\varepsilon}(u_2, v_2) \setminus \sigma_{\varepsilon/\varepsilon}(u_1, v_1)| \cdot max(0, |\gamma(u_2) - 1|) \end{aligned}$$

Intuitively, the gain of an edge, representing two similar transitions, is determined by the number of alphabet symbols used in both transitions that do not access the stack ($\sigma_{\varepsilon/\varepsilon}$). The gain is also negatively influenced by the number of symbols that are specific to one of the transitions. If a symbol is used exclusively within one of the transitions that does not access the stack, then this transition needs to be replaced with transitions that have the same pop and push symbols for each stack symbol that can occur in this transition (the source state of the transition). This replacement ensures that the other transitions are prevented from performing a step over this unique symbol. This expansion is illustrated in Figure 4.4 between two original transitions from state s_1 to s_3 and s_2 to s_4. In this case, the transition over the symbol b has to be expanded into two transitions, using stack symbols 1 and 3.

Definition 4.4. *Let $P = (V, E)$ be a subgraph of a superproduct.* ***Vertex gain*** *is a function $G_v : V \to \mathbb{Z}$ defined as $G_v(u) = G_e(u, u)$.*

The calculation of gains for all subgraphs is the main bottleneck of our algorithm. The calculation of the gain for a single rooted subgraph is done in $\mathcal{O}(|V_{\hat{r}}| + |E_{\hat{r}}|)$ time. Since each node can be almost fully connected with all other nodes, then the gain is calculated in $\mathcal{O}(|V_{\hat{r}}|^2)$. Consequently, the gains for all subgraphs are computed in $\mathcal{O}(|V|^3)$ time. As the superproduct is formed from the automaton by pairing all combinations of states, we get $\mathcal{O}(|V|) = \mathcal{O}(|Q|^2)$. Therefore, the calculation of gains for all subgraphs takes $\mathcal{O}(|Q|^6)$ time. This can make our reduction method inefficient for large automata.

However, complexity does not appear to be a significant issue for automata with a reasonable size (up to hundreds of states). Moreover, the primary objective of this work was to serve as a proof of concept rather than provide the optimal algorithm.

To optimize the computation time of gains, we can specify the maximal distance d to which the calculation of root gains is computed. Further optimizations can be achieved by selecting only a subset of states for which the superproduct has to be calculated, or by selecting only a subset of superproduct vertices (roots) for which we want to know the gains.

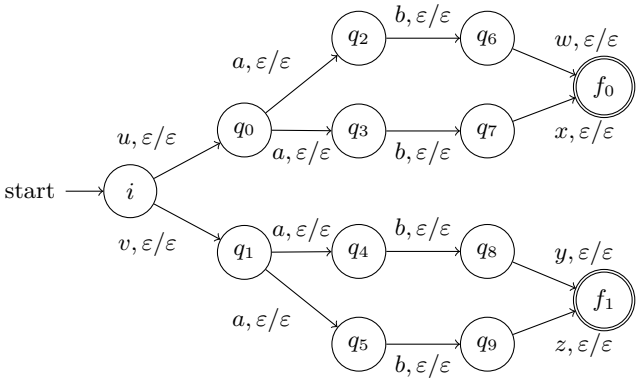
4.3 Only Linear Subgraphs

Not every rooted subgraph of a superproduct is good for procedure creation. It is crucial to select only linear subgraphs because each branching requires transitions that rename old stack symbols to new ones, for distinguishing between different branches. The linear subgraph is obtained using the *forest* function, which breaks down the subgraph with multiple branches into a set of linear subgraphs while maintaining the maximal gain.

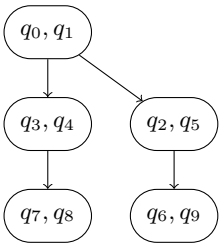
Definition 4.5. *For a given subgraph $SP_{\hat{r}}(M) = (V_{\hat{r}}, E_{\hat{r}})$, the function* ***forest****, is defined as forest* : $2^{V_{\hat{r}}} \times 2^{E_{\hat{r}}} \to 2^{2^{V_{\hat{r}}} \times 2^{E_{\hat{r}}}}$. *The forest function returns a set of procedure candidates $P_0 = (V_0, E_0)$, $P_1 = (V_1, E_1), \ldots, P_n = (V_n, E_n)$, where the following conditions hold:*

1. $V_{\hat{r}} = \bigcup_{0 \leq k \leq n} V_k$,
2. $E_{\hat{r}} \supseteq \bigcup_{0 \leq k \leq n} E_k$,
3. $|E_{\hat{r}}| - n = |\bigcup_{0 \leq k \leq n} E_k|$,
4. $\forall 0 \leq k < l \leq n : V_k \cap V_l = \emptyset \wedge E_k \cap E_l = \emptyset$, *and*
5. $\sum_{0 \leq k \leq n} G(P_k)$ *is maximal.*

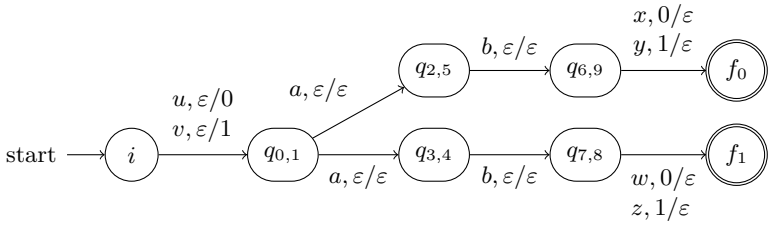
The main reason behind the need for a linear structure is to be able to distinguish between different branches of the procedure. The branching occurs at vertices with more successors. The branching is similar to forking a process, with one branch maintaining its original stack symbols (the parent process maintaining its pid), while other branches start using new stack symbols (children processes receiving new pids) to distinguish themselves. If the function *forest* and new stack symbols were not used, further reduction and creation of new procedures by combining similar procedures would not be possible because we would not be able to distinguish between different branches of procedures. Although this can work in some cases (see Figure 4.2), it does not work in general (see Figure 4.3).



An NPDA$_1$ $M_{4.2}$ with similar transitions over ab.

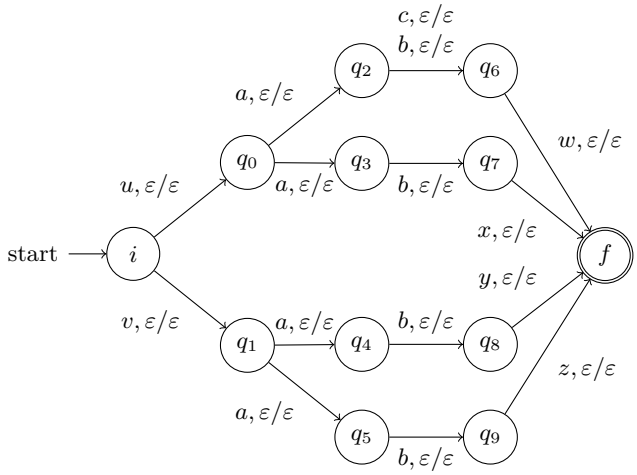


Rooted subgraph $SP_{\widehat{(q_0,q_1)}}(M_{4.2})$.

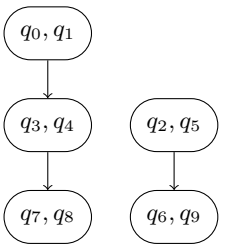


The resulting automaton after the procedure creation that was performed based on the subgraph $SP_{\widehat{(q_0,q_1)}}(M_{4.2})$ with a tree structure.

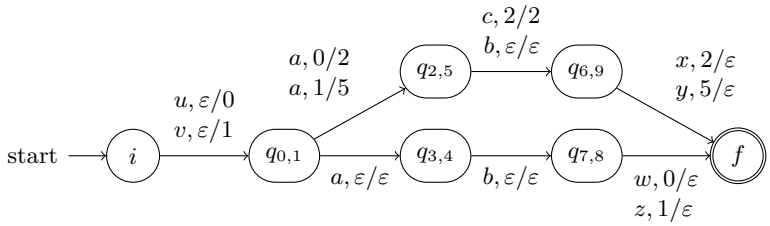
Figure 4.2: In this figure, we can see an example where it is not necessary to create a new stack symbol at the branching point (state $q_{0,1}$). However, this practice cannot be applied in general, and therefore we must ensure that each procedure candidate has a linear structure.



An NPDA $M_{4.3}$ with similar transitions over ab.



Subgraph $forest(SP_{\widehat{(q_0,q_1)}}(M_{4.3}))$



The resulting automaton after we created a procedure based on two procedure candidates obtained by forestification of the rooted subgraph $SP_{\widehat{(q_0,q_1)}}(M_{4.3})$.

Figure 4.3: In this figure, we can see a correct procedure construction, that uses new stack symbols for new branches, based on the set of procedure candidates (with a linear structure, as the condition requires). At the branching point (state $q_{0,1}$), a new stack symbols 2 and 5 are used to determine the branch. Using the new symbol, we can further reduce branches by creating a procedure for a word b. This would not be possible if we did not use the new stack symbol but instead reused the symbol 0, because we would lack a way to specify that the transition c is part of the upper branch.

4.4 Procedure Candidate

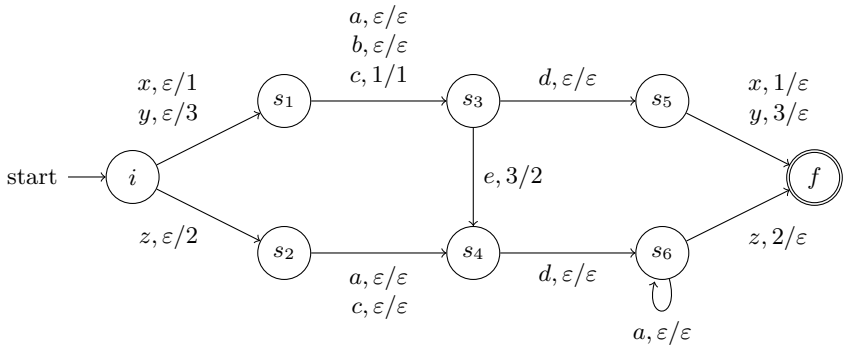
A procedure candidate is an oriented, connected linear subgraph of a superproduct. It carries information on how to construct a procedure and indicates a reduction in the number of transitions. Vertices in a procedure candidate represent pairs of states that have similar transitions, while the edges represent these similar transitions.

Definition 4.6. *Let M be an NPDA$_1$. A **procedure candidate** for an automaton M is an oriented connected linear graph $P = (V \subseteq Q \times Q, E \subseteq V \times V)$ following the conditions:*

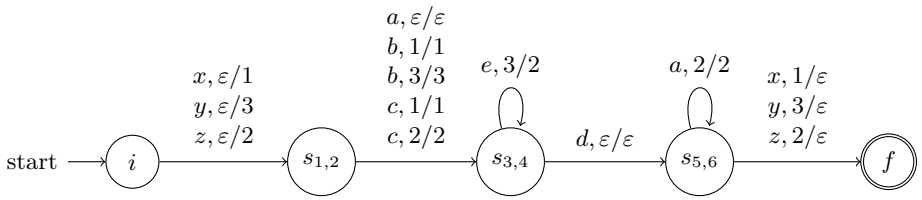
1. $\forall (u_1, u_2), (v_1, v_2) \in V : (u_1, u_2) = (v_1, v_2) \vee |\{u_1, u_2, v_1, v_2\}| = 4$
2. $|E| = |V| - 1$
3. $\forall u, v \in V : (u = v \implies (u, v) \notin E) \wedge ((u, v) \in E \implies \nexists x \in V : x \neq v \wedge (u, x) \in E)$
4. $\forall ((u_1, u_2), (v_1, v_2)) \in E : \sigma_{\varepsilon/\varepsilon}(u_1, v_1) \cap \sigma_{\varepsilon/\varepsilon}(u_2, v_2) \neq \emptyset$

Intuitively, the first condition of a procedure candidate ensures that no vertex contains the same automaton state multiple times. This condition is achieved by applying the *RemoveReuse* algorithm. The second and third conditions ensure that the graph is both connected and linear. This structure of a procedure candidate is obtained by the function *forest*. The final condition guarantees that each edge of the procedure candidate represents similar transitions. To evaluate the reduction potential of a procedure candidate, we use gain functions for both edges and vertices.

Definition 4.7. ***Procedure candidate gain*** *of a procedure candidate* $P = (V, E)$ *is a function* $G : P \to \mathbb{Z}$ *defined as* $G(P) = \sum_{e \in E} G_e(e) + \sum_{v \in V} G_v(v)$.



NPDA$_1$ $M_{4.4}$ with two similar transition sequences specified by a procedure candidate $P_{4.4} = (\{s_{1,2} \coloneqq (s_1, s_2), s_{3,4} \coloneqq (s_3, s_4), s_{5,6} \coloneqq (s_5, s_6)\}, \{(s_{1,2}, s_{3,4}), (s_{3,4}, s_{5,6})\})$ with a gain of 1.



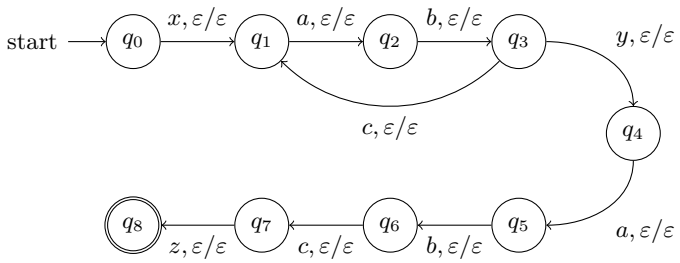
Automaton $M_{4.4}$ after creating procedure according to the procedure candidate $P_{4.4}$.

Figure 4.4: This figure illustrates the creation of a procedure based on the procedure candidate. It also demonstrates how the procedure construction must create new transitions for some transitions with the push and pop symbols set to ε to preserve the automaton's language. Furthermore, it highlights the importance of considering the gain of the procedure candidate to determine if the construction is worthwhile.

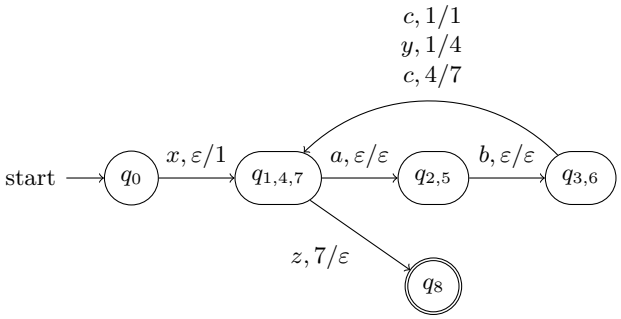
The reason for prohibiting multiple occurrences of the same state through the procedure candidate vertices (as stated by the first condition) is illustrated in Figure 4.5. When state q_1 is reused in the subgraph representing the procedure candidate, it implies a cycle in the first sequence of transitions but not in the second sequence. As a result, the edge between the vertices (q_3, q_5) and (q_1, q_7) does not represent a transition $c, \varepsilon/\varepsilon$, but rather a transitions $c, 1/1$ and $c, 4/7$. This led to misleading information about the gain of a procedure candidate, promising a higher gain than it was able to provide.

To address this issue, it is crucial to ensure that the subgraph representing the procedure candidate, derived from the rooted subgraph, does not reuse any automaton states. To enforce the condition of unique automaton states across vertices in a rooted subgraph and maximize its gain, we iteratively select vertices with states that have not been used

before, along with their successors. We prioritize those with the highest gain in their rooted subgraph. This approach is implemented in the *RemoveReuse* algorithm.



An NPDA $M_{4.5}$ with superproduct consisting of identity nodes and two subgraphs over vertices $(q_1, q_4), (q_2, q_5), (q_3, q_6), (q_1, q_7)$ and $(q_4, q_1), (q_5, q_2), (q_6, q_3), (q_7, q_1)$ respectively.



Resulting automaton with procedure created based on the subgraph over the vertices $(q_1, q_4), (q_2, q_5), (q_3, q_5), (q_1, q_7)$, that reuses the state q_1.

Figure 4.5: The reuse of the state q_1 provides misleading information for the creation of a procedure. According to the definition, there should be three transition that does not use stack, but in this case, there are only two. Moreover, a special transition $c, 4/7$ is needed to enforce that after this transition, the cycle will not be entered again.

Algorithm 1: *RemoveReuse*

Input: A rooted subgraph $SP_{\hat{r}}(M) = (V_{\hat{r}} \subseteq Q \times Q, E_{\hat{r}} \subseteq V_{\hat{r}} \times V_{\hat{r}})$.
Output: A subgraph $(V \subseteq V_{\hat{r}}, E \subseteq E_{\hat{r}})$ with possibly maximal gain.

```
1  V ↤ ∅
2  visited ↤ ∅
3  worklist ↤ Stack()
4  worklist.push(r)
5  while wordlist ≠ ∅ do
6      uv₁ ↤ worklist.top()
7      (u₁, v₁) ↤ uv₁
8      if {u₁, v₁} ∩ visited ≠ ∅ then
9          continue
10     V ↤ V ∪ {uv₁}
11     succ ↤ {uv₂ ∈ V_r̂ | (uv₁, uv₂) ∈ E_r̂}
12     forall (uv₂) ∈ asc_sorted(succ, by = λx.G(SP_x̂(M))) do
13         worklist.push(uv₂)
14 E ↤ V × V ∩ E_r̂
15 return (V, E)
```

The algorithm *RemoveReuse* traverses the subgraph $SP_{\hat{r}}(M)$ using DFS (implemented with a stack), starting from the root r. The successors of the currently examined vertex are not pushed onto the stack in random order. Instead, they are pushed in ascending order by the gains of their rooted subgraph, ensuring that the vertex with the highest gain is on the top of the stack. Alternatively, the traversal approach can be changed to BFS by using a queue on line 3 and descending sort on line 12. Although we have decided to choose DFS, similar results can be achieved with BFS.

By selecting the rooted subgraph of a superproduct with the highest gain and applying the *RemoveReuse* algorithm and the *forest* function, we obtain a procedure candidate. There are five types of transitions in the original automaton that are associated with the procedure candidate, namely:

1. Call transitions that enter states used within the procedure candidate,
2. Return transitions that exit states used within the procedure candidate,
3. Core transitions that represent similar transitions within the procedure candidate,
4. Guard transitions specific to one substructure, and
5. Switch transitions that interconnect different similar substructures.

In the following text, we consider an NPDA$_1$ denoted as $M = (Q, \Sigma, \Gamma\varepsilon, \delta, I, \lambda, F, \phi)$ and a procedure candidate denoted as $P = (V, E)$.

The projection operation on the i-th element, often denoted by π_i, is used to access the element x_i of a tuple. For a tuple $x = (x_1, \ldots, x_i, \ldots, x_n) \in X_1 \times \cdots \times X_i \times \cdots \times X_n$, the projection of the i-th element of x is $\pi_i(x) = x_i$. This notation can be generalized to a subset of a Cartesian product to access the i-th element of each tuple in the set. For a given subset $X \subseteq X_1 \times \cdots \times X_i \times \cdots \times X_n$, the projection of the i-th element of each tuple from X is the set $\pi_i(X) = \{\pi_i(x) \in X_i \,|\, x \in X\}$.

Definition 4.8. ***Procedure candidate states*** *are given by a set of states used within procedure vertices, defined as* $Q(P) = \pi_1(V) \cup \pi_2(V)$.

We use the projection of the set of tuples to define the set of original procedure candidate states, which contains all the states used in the procedure candidate vertices. These states will be used to classify transitions.

Definition 4.9. ***Call transitions****, for M and P, are defined by a set* $\delta_{call}(M, P) = \{(q, a, \alpha, \beta, r) \,|\, (r, \beta) \in \delta(q, a, \alpha),$ *where* $q \notin Q(P), r \in Q(P), a \in \Sigma, \alpha, \beta \in \Gamma_\varepsilon\}$.

Call transitions specify transitions that enter the procedure candidate. The source state of such a transition must be outside the current procedure candidate. However, the source state has to be in the procedure candidate. For example, $\delta_{\text{call}}(M_{4.4}, P_{4.4}) = \{(i, x, \varepsilon, 1, s_1), (i, y, \varepsilon, 3, s_1), (i, z, \varepsilon, 2, s_2)\}$.

Definition 4.10. ***Return transitions****, for M and P, are defined by a set* $\delta_{ret}(M, P) = \{(q, a, \alpha, \beta, r) \,|\, (r, \beta) \in \delta(q, a, \alpha),$ *where* $q \in Q(P), r \notin Q(P), a \in \Sigma, \alpha, \beta \in \Gamma_\varepsilon\}$.

The set of return transitions is the opposite of the set of call transitions. It contains transitions that exit the procedure candidate. The target state of the return transition must be outside the current procedure candidate, but it can be part of some procedure. For example, $\delta_{\text{ret}}(M_{4.4}, P_{4.4}) = \{(s_5, x, 1, \varepsilon, f), (s_5, y, 3, \varepsilon, f), (s_5, z, 2, \varepsilon, f)\}$.

Definition 4.11. ***Core transitions****, for M and P, are defined by a set* $\delta(M, P) = \{(q, a, \varepsilon, \varepsilon, r) \,|\, (r, \varepsilon) \in \delta(q, a, \varepsilon) \wedge \exists (q, q'), (r, r') \in V : (r', \varepsilon) \in \delta(q', a, \varepsilon),$ *where* $a \in \Sigma\}$.

Core transitions are the most important transitions in the procedure candidate because only those positively contribute to its gain. A core transition is a transition within procedure candidate states that exists in both automaton substructures, indicated by an edge, and can therefore be represented just once. For example, $\delta_{\text{core}}(M_{4.4}, P_{4.4}) = \{(s_1, a, \varepsilon, \varepsilon, s_3), (s_2, a, \varepsilon, \varepsilon, s_4), (s_3, d, \varepsilon, \varepsilon, s_5), (s_4, d, \varepsilon, \varepsilon, s_6)\}$.

Definition 4.12. ***Guard transitions****, for M and P, are defined by a set $\delta_{guard}(M, P) = \{(q, a, \alpha, \alpha, r) \mid (r, \alpha) \in \delta(q, a, \alpha) \wedge \nexists q', r' \in P(Q) : (r', \alpha) \in \delta(q', a, \alpha) \wedge (((q, q'), (r, r')) \in E \vee ((q', q), (r', r)) \in E)$, where $a \in \Sigma$, $\alpha \in \Gamma_\varepsilon\}$.*

Guard transitions also exist within the procedure candidate states, represented by an edge, but they are specific to only one of the substructure. For example, $\delta_{guard}(M_{4.4}, P_{4.4}) = \{(s_1, b, \varepsilon, \varepsilon, s_3), (s_1, c, \varepsilon, \varepsilon, s_3), (s_2, c, \varepsilon, \varepsilon, s_4), (s_5, a, \varepsilon, \varepsilon, s_5)\}$.

Definition 4.13. ***Switch transitions****, for M and P, are defined by a set $\delta_{aux}(M, P) = \{(q, a, \alpha, \beta, r) \mid (r, \beta) \in \delta(q, a, \alpha) \wedge q \in \pi_1(V) \iff r \in \pi_2(V)$, where $a \in \Sigma$, $\alpha, \beta \in \Gamma_\varepsilon\}$.*

The set of switch transitions is a set of transitions within the procedure candidate states that are not represented by an edge. For example $\delta_{aux}(M_{4.4}, P_{4.4}) = \{(s_3, e, 3, 2, s_4)\}$.

4.5 Measuring States Reduction vs Transitions Reduction

At the end of this chapter, it is important to note how to measure the reduction achieved by transforming the input NFA into NPDA$_1$. The reduction cannot be measured solely by a decrease in the number of states. We must also consider the size of the stack alphabet and the number of transitions. This is because all the information provided by the states in the NFA can be transferred to the stack in the NPDA$_1$. Although the resulting automaton has a single state, this reduction does not provide any benefit, as the stack alphabet size is equal to the number of states in the original automaton, and the number of transitions does not decrease. To achieve precise measurements of the state reduction, it is necessary to compare the number of states in the original NFA with the sum of the number of states and the size of the stack alphabet in the resulting NPDA$_1$. Unlike the issue with the states reduction, the number of transitions in the resulting NPDA$_1$ can be directly compared to the number of transitions in the original NFA.

It may be apparent that each NFA can be easily transformed into an NPDA$_1$ employing a single state. Such a reduction is essentially useless because the resulting automaton contains the same amount of transitions as the original, and the stack alphabet has the size of the state set in the original automaton. An example is shown in Figure 4.6.

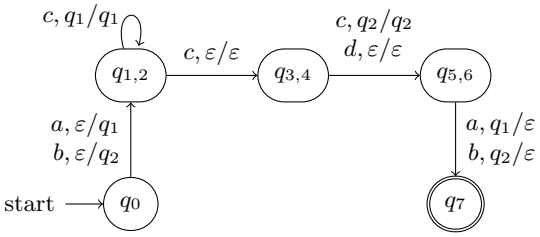
Theorem 4.1. *Every NFA can be converted into an NPDA$_1$ with a single state.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ be an NFA. The there is language equivalent automaton NPDA$_1$ $B = (Q_B \coloneqq \{q\}, \Sigma, \Gamma, \delta_B, Q_B, \lambda, Q_B, \phi)$ with its elements defined as follows:

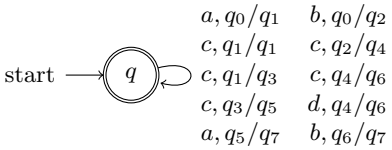
- $\Gamma = Q_A$,
- $\delta_B(q, a, \alpha) = (q, \beta) \iff \beta \in \delta_A(\alpha, a)$,
- $\lambda(q) = I_A$,
- $\phi(q) = F_A$.

□

The prior proof highlights the ease with which the number of automaton states can be reduced by using the stack to retain information about the current state of computation. As the result, the automaton has the stack alphabet size equal to the number of states in the original automaton.



The optimally converted automaton $M_{4.1}$ from Figure 4.1.



The NPDA$_1$, with a single state, created from $M_{4.1}$, represents the least effective reduction approach.

Figure 4.6: This figure illustrates the contrast between an optimal and inefficient conversion of the NFA into an NPDA$_1$. The optimal NPDA$_1$ consists of 5 states, 8 transitions, and utilizes 2 stack symbols. In contrast, the suboptimal NPDA$_1$, despite having only 1 state, consists of 9 transitions and requires 8 stack symbols, resulting in no reduction.

Chapter 5

Procedure Mapping Reduction

At this point, we understand how to identify the best possible procedure candidates. What remains is understanding how to construct procedures and integrate them into the automaton. This chapter describes the algorithm that transforms input NFA into NPDA$_1$ by creating procedures based on procedure candidates that represent similar transition sequences, with the aim of achieving a maximum reduction in the size of the automaton. First, the main algorithm guiding the transformation is presented, followed by a section dedicated to mapping the procedure candidates. Finally, at the end of the chapter, two algorithms are presented: one for reducing the size of the stack alphabet and another for reducing the number of unnecessary guard transitions by replacing them with core transitions.

5.1 Main Reduction Algorithm

This section presents the main loop of the reduction algorithm that converts the input NFA into NPDA$_1$ using the Procedure mapping. The algorithm takes an NFA M as the input and an optional argument d, which specifies the maximum depth at which the gain of the rooted subgraphs of the superproduct is calculated. The argument d is used primarily for automata with a large number of states, where the calculation of the gain of the rooted subgraphs of the superproduct is computationally expensive.

Algorithm 2: Reduction of NFA using Procedure mapping.

Input: NFA $A = (Q_{in}, \Sigma, \delta_{in}, I_{in}, F_{in})$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1  B ↤ toNPDA_1(A)                                          // see Theorem 2.1
2  SP ↤ SP(B)
3  while ∃r ∈ Q : 0 < G(SP_r̂) ∧ ∀q ∈ Q : G(SP_q̂) ≤ G(SP_r̂) do
4      PP ↤ forest(unique(SP_r̂))
5      forall P ∈ PP do
6          B ↤ mapProcedure(B, P)
7          Q ↤ Q \ Q(P)
8      SP ↤ SP(B)                    // update can be done only locally based on the PP
9  B ↤ guardReduction(stackReduction(B))
10 return B
```

The algorithm begins by transforming the input NFA into an equivalent NPDA$_1$, following the same principle as described in the second part of the proof of Theorem 2.1.

Subsequently, the algorithm calculates the superproduct of the automaton. Afterward, a rooted subgraph with the highest positive gain is obtained from the superproduct. Then, it is transformed into a set of procedure candidates through the application of the algorithm *RemoveReuse* followed by the *forest* function. The algorithm then iterates over the set of procedure candidates, creating a procedure for each candidate and integrating it into the automaton. Upon integration of a procedure, all original states of the automaton that have been utilized within the procedure candidate are removed, as their behaviors are already represented by the procedure. At the end of each iteration, the superproduct is updated. Instead of recalculating the entire superproduct, it can be updated with respect to the new states that have been created along with the procedure. The algorithm terminates when there is no procedure candidate in the superproduct with a positive gain. Finally, the resulting NPDA_1 automaton is returned.

5.2 Procedure Mapping

After Algorithm 2 has selected the procedure candidate $P = (V, E)$ for mapping, the following algorithm will be used to create new states for the procedure and to map all transitions from the states used within the procedure candidate into the procedure. This includes call, return, core, guard, and switch transitions. Additionally, the algorithm will map the initial and final properties of the states.

During mapping of transitions, it is crucial to track the stack symbols used within each state. To achieve this, we employ a function $\Gamma_{proc} : Q(P) \to 2^{\Gamma_\varepsilon}$, which maps the original states used in the procedure candidate to the symbols that can appear on the top of the stack in those states. Additionally, it is necessary to determine which procedure states represent the original states. This is accomplished by a function $Q_{proc} : Q(P) \to Q$ that associates the original states with the newly created procedure states. Furthermore, we will use the notation $pick(S)$ to denote an arbitrary element from the set S, and $root(P)$ to denote the root vertex of the procedure candidate P.

Algorithm 3: mapProcedure

Input: NPDA_1 $A = (Q_{in}, \Sigma, \Gamma_{in}, \delta_{in}, I_{in}, \lambda_{in}, F_{in}, \phi_{in})$ and a procedure candidate $P = (V, E)$
Output: NPDA_1 $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(N) \equiv L(M)$

$B \leftarrow\!\shortmid A$

`// Determine stack symbols`
$(r_1, r_2) \leftarrow\!\shortmid root(P)$
$\Gamma_{proc} : Q(P) \to 2^{\Gamma_\varepsilon}$
$\Gamma_{proc}(r_1) \leftarrow\!\shortmid \{r_1\}$ **if** $\gamma(r_1) = \{\varepsilon\}$ **else** $\gamma(r_1)$
$\Gamma_{proc}(r_2) \leftarrow\!\shortmid \{r_2\}$ **if** $\gamma(r_2) = \{\varepsilon\}$ **else** $\gamma(r_2)$
$\Gamma_\varepsilon \leftarrow\!\shortmid \Gamma_\varepsilon \cup \Gamma_{proc}(r_1) \cup \Gamma_{proc}(r_2)$

`// Create procedure states`
$Q_{proc} : Q(P) \to Q$
forall $(u, v) \in V$ **do**
    $uv \leftarrow\!\shortmid B.newState()$ `//` $uv \notin Q$
    $Q \leftarrow\!\shortmid Q \cup \{uv\}$
    $Q_{proc}(u) \leftarrow\!\shortmid uv$
    $Q_{proc}(v) \leftarrow\!\shortmid uv$
    $\Gamma_{proc}(u) \leftarrow\!\shortmid \Gamma_{proc}(r_1)$
    $\Gamma_{proc}(v) \leftarrow\!\shortmid \Gamma_{proc}(r_2)$
    $\Gamma_{proc}(uv) \leftarrow\!\shortmid \Gamma_{proc}(u) \cup \Gamma_{proc}(v)$

```
   // Map procedure
14 B ↤ mapCall(B, δ_call(A, P), Γ_proc, Q_proc)
15 B ↤ mapReturn(B, δ_call(A, P), Γ_proc, Q_proc)
16 B ↤ mapCore(B, δ_call(A, P), Q_proc)
17 B ↤ mapGuard(B, δ_call(A, P), Γ_proc, Q_proc)
18 B ↤ mapSwitch(B, δ_call(A, P), Γ_proc, Q_proc)
19 B ↤ mapInitial(B, P, Γ_proc(r_1), Γ_proc(r_2))
20 B ↤ mapFinal(B, P, Γ_proc(r_1), Γ_proc(r_2))
21 return B
```

The algorithm takes an NPDA_1 and a procedure candidate as input and returns a modified automaton with a procedure created based on the given procedure candidate. First, a copy B of the input automaton is made. Then, a function Γ_{proc} mapping states to stack symbols is established. Each root state is assigned a unique stack symbol if it does not already have one. These stack symbols will be used to determine which procedure transition belongs to which original transition sequence. After establishing the stack symbols for each root state, they are redistributed to their successors that are used within a procedure candidate. The algorithm also creates new states for the procedure and associates them with their original states using the function Q_{proc}. Finally, mapping of all types of transitions, as well as initial and final states, is performed.

5.2.1 Call Transitions

The *mapCall* algorithm takes four parameters as input: NPDA_1, a set of call transitions T, a stack function Γ_{proc} that specifies stack symbols for each state of the procedure candidate, and a mapping function Q_{proc} that maps the original states to the corresponding newly created states of the procedure. The algorithm maps all call transitions from the set T that enter one of the procedure candidate states to the associated procedure states given by the mapping function Q_{proc}. If the target state r of a transition has already been part of some procedure ($1 < \Gamma_{proc}(r)$), the push symbol β is left unchanged. Otherwise, the push symbol is determined by the stack function Γ_{proc}. At the end of the algorithm, the automaton with mapped call transitions is returned.

Algorithm 4: mapCall

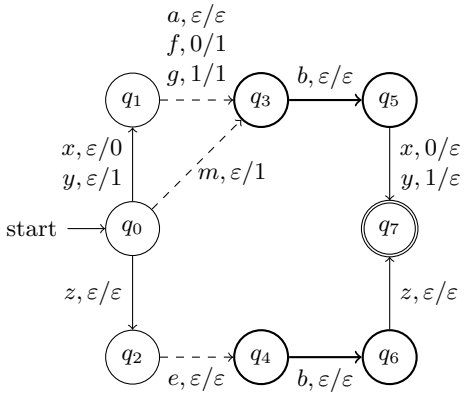
Input: NPDA_1 $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$,
set of call transitions $T \subseteq (Q \times \Sigma \times \Gamma_\varepsilon \times \Gamma_\varepsilon \times Q)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA_1 $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1  B ↤ A
2  forall (q, a, α, β, r) ∈ T do
3      r' ↤ Q_proc(r)
4      if |Γ_prov(r)| = 1 then                                  // new stack symbol
5          assert(β = ε)
6          δ(q, a, α) ↤ δ(q, a, α) ∪ {(r', pick(Γ_proc(r)))}
7      else                                                     // copy stack symbol
8          assert(1 < |Γ_proc(q)| ∨ β ≠ ε)
9          δ(q, a, α) ↤ δ(q, a, α) ∪ {(r', β)}
10 return B
```

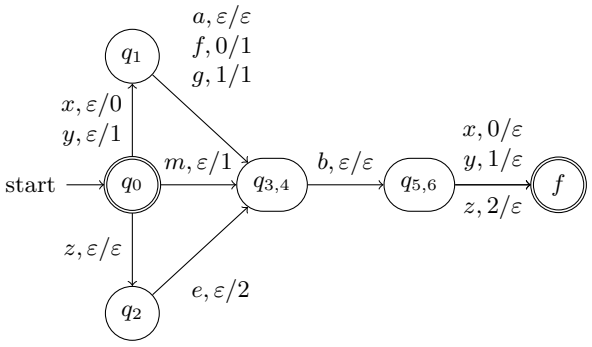
$\Gamma_{proc}(q) = 1$	$\Gamma_{proc}(r) = 1$	$\beta = \varepsilon$	Is possible	β'
✗	✗	✗	yes	β
✗	✗	✓	yes	β
✗	✓	✗	no	$pick(\Gamma_{proc}(r))$
✗	✓	✓	yes	$pick(\Gamma_{proc}(r))$
✓	✗	✗	yes	β
✓	✗	✓	no	β
✓	✓	✗	no	$pick(\Gamma_{proc}(r))$
✓	✓	✓	yes	$pick(\Gamma_{proc}(r))$

Table 5.1: Possible combinations of the behavior of source and target states (whether they are part of the procedure, $1 < \Gamma_{proc}(s)$, or not, $\Gamma_{proc}(s) = 1$) and the symbol being pushed onto the stack (if it is specified or is ε) during the call transitions. The column *Is possible* states whether the call transition with this combination can exist in NPDA_1. The β' column specifies the modified push symbol that will be used in the mapped transition.

When mapping a call transition, five combinations of the source and the target state behavior and the push symbol are valid. Table 5.1 shows all possible combinations. The third combination is not valid because it represents a return from a procedure to a state r that is not part of any procedure, but specifies the push symbol. This push does not make sense because there is nothing for the automaton to remember. The sixth combination is also not valid because it denotes a transition that enters an already existing procedure but does not specify the push symbol (the automaton will not know where to return at the end of the procedure). The seventh combination is not valid because it means that a transition between two states, where neither of them is part of any procedure, specifies the push symbol. This push is as invalid as the one in the third combination. It can be seen that, for valid combinations (all valid combinations can be seen in Figure 5.1), the modified push symbol β' depends only on the condition whether the target state r is part of some procedure ($1 < \Gamma_{proc}(r)$) or not ($\Gamma_{proc}(r) = 1$).



Automaton $M_{5.1}$ with a procedure candidate over bold transitions and states and dashed call transitions.



The resulting automaton $M_{5.1}$ after the mapping of the call transitions.

Figure 5.1: This Figure illustrates the mapping of different call transitions (dashed lines) in the automaton $M_{5.1}$, according to *mapCall* algorithm.

5.2.2 Return Transitions

The *mapReturn* algorithm takes four parameters as input: NPDA_1, the set of return transitions T, the stack function Γ_{proc}, which specifies the stack symbols for each state of the procedure candidate, and the mapping function Q_{proc}, which maps the original states to the corresponding newly created states of the procedure. The algorithm maps all return transitions from T that exit the procedure candidate states that are associated with procedure states by the mapping function Q_{proc}. The algorithm distinguishes three situations. First, if the source and target states are part of the procedure and the pop and push symbols are set to ε (meaning that this transition simplifies many transitions with the same alphabet symbol but with different stack symbols), the algorithm must unwind this comprehension and create a guard transition for each stack symbol that this transition represents. Second, if the source state is part of the procedure and the pop symbol is different from ε (the only valid combination), then the algorithm preserves the pop symbol. Lastly, if the source state is not part of any procedure and the pop symbol is set to ε, the algorithm uses a newly created stack symbol as the pop symbol. At the end of the algorithm, the automaton with mapped return transitions is returned.

Algorithm 5: mapReturn

Input: NPDA_1 $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$,
set of return transitions $T \subseteq (Q \times \Sigma \times \Gamma_\varepsilon \times \Gamma_\varepsilon \times Q)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA_1 $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

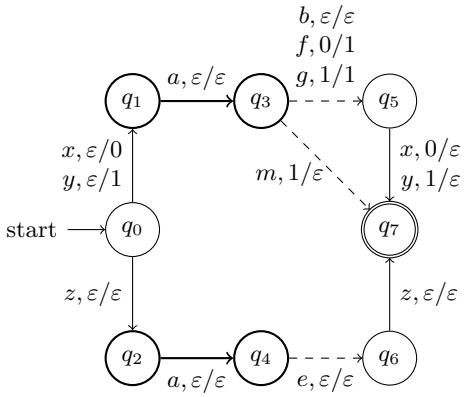
```
1  B ↤ A
2  forall (q, a, α, β, r) ∈ T do
3      q' ↤ Q_proc(q)
4      if 1 < |Γ_proc(q)| ∧ α = ε ∧ β = ε then          // expand all stack symbols
5          assert(1 < |Γ_proc(r)|)
6          forall α' ∈ Γ_proc(q) do
7              δ(q', a, ε) ↤ δ(q', a, α') ∪ {(r, α')}
8      else if 1 < |Γ_proc(q)| then                     // copy stack symbol
9          assert(α ≠ ε)
10         δ(q', a, α) ↤ δ(q', a, α) ∪ {(r, β)}
11     else                                             // new stack symbol
12         assert(α = ε)
13         α' ↤ pick(Γ_proc(q))
14         δ(q', a, α') ↤ δ(q', a, α') ∪ {(r, β)}
15 return B
```

The mapping of return transitions is more complex than that of call transitions. Within this mapping, five valid combinations of source state behavior, pop, and push symbols can occur, all depicted in Figure 5.2. Table 5.2 presents all theoretically possible combinations but highlights only five of them, which are semantically valid in the context of NPDA_1. All valid combinations have been discussed in the description of Algorithm 5. However, let us also consider the impossible combinations. The third combination in the table describes a situation where a state, that is part of some procedure, pushes a symbol onto the stack without a previous pop. This is not possible since NPDA_1 can contain a maximum of one symbol on the stack. The fifth combination describes a situation where a state that is not part of any procedure applies pop and push to the stack. However, the stack is empty for states outside procedures. The sixth combination illustrates a similar concept as the fifth

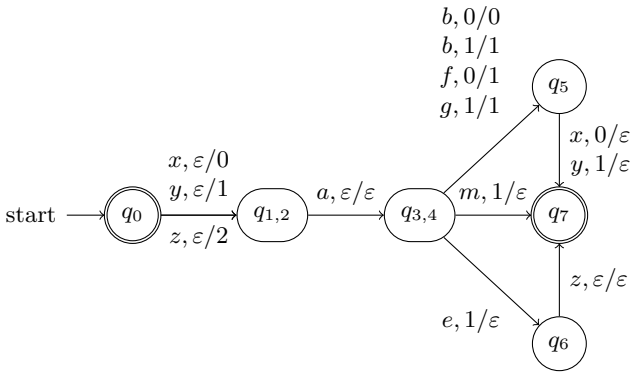
combination. A state outside a procedure cannot apply pop to the stack. As shown in the table, the modified mapped pop symbol α' depends on the condition whether the source state q is part of some procedure ($1 < \Gamma_{proc}(q)$) or not ($\Gamma_{proc}(q) = 1$), with an exception for the fourth combination, where the pop and push symbols have to be tested.

$\Gamma_{proc}(q) = 1$	$\alpha = \varepsilon$	$\beta = \varepsilon$	Is possible	α'
✗	✗	✗	yes	α
✗	✗	✓	yes	α
✗	✓	✗	no	α
✗	✓	✓	yes	$\forall \alpha' \in \Gamma_{proc}(q)$
✓	✗	✗	no	$pick(\Gamma_{proc}(q))$
✓	✗	✓	no	$pick(\Gamma_{proc}(q))$
✓	✓	✗	yes	$pick(\Gamma_{proc}(q))$
✓	✓	✓	yes	$pick(\Gamma_{proc}(q))$

Table 5.2: Possible combinations of the behavior of the source state (whether it is part of the procedure, $1 < \Gamma_{proc}(q)$, or not, $\Gamma_{proc}(q) = 1$) and the pop and push symbols (whether they are specified or set to ε) for the return transitions. The column *Is possible* indicates whether the return transition with this combination can exist in NPDA_1. The α' column specifies the modified pop symbol that will be used on the mapped transition.



Automaton $M_{5.2}$ with procedure candidate over bold transitions and states and dashed return transitions.



The resulting automaton $M_{5.2}$ after the mapping of the return transitions.

Figure 5.2: This Figure illustrates mapping of different return transitions (dashed transitions) in the automaton $M_{5.2}$ according to *mapReturn* algorithm.

5.2.3 Core Transitions

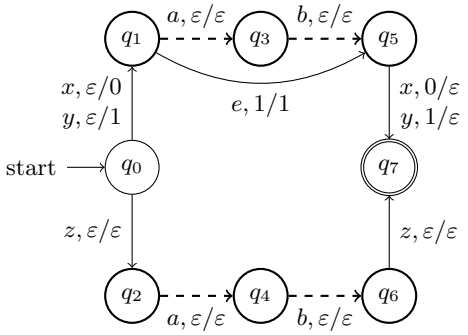
The algorithm *mapCore* takes three parameters as input: the input NPDA_1, the set of core transitions T, and the mapping function Q_{proc}, which maps original states to the corresponding procedure states. The algorithm maps all core transitions from T to the procedure states given by the mapping function Q_{proc}. As you can see, the algorithm does not transform pop or push symbols. This straightforward behavior is due to Definition 4.11, which specifies that core transitions do not operate with the stack (pop and push are

set to ε). Therefore, these transitions remain unchanged, and we only need to map the source and target states to the corresponding procedure states using the function Q_{proc}. In the end, the resulting automaton with mapped core transitions is returned.

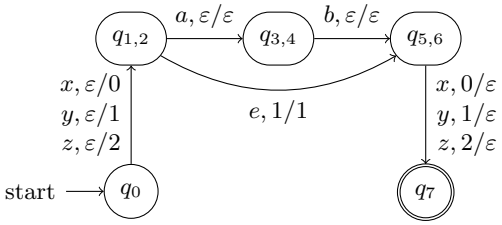
Algorithm 6: mapCore

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$,
set of return transitions $T \subseteq (Q \times \Sigma \times \Gamma_\varepsilon \times \Gamma_\varepsilon \times Q)$, and
state function $Q_{proc} : Q(P) \to Q$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1 B ↤ A
2 forall (q, a, α, β, r) ∈ T do
3     assert(α = ε ∧ β = ε)
4     q' ↤ Q_proc(q)
5     r' ↤ Q_proc(r)
6     δ(q', a, ε) ↤ δ(q', a, ε) ∪ {(r', ε)}
7 return B
```



Automaton $M_{5.3}$ with procedure candidate over bold transitions and states and dashed bold core transitions.



The resulting automaton $M_{5.3}$ after the mapping of the core transitions.

Figure 5.3: This figure illustrates the mapping of core transitions (dashed bold transitions) in the automaton $M_{5.3}$ according to *mapCore* algorithm.

5.2.4 Guard Transitions

Although the *mapGuar* algorithm maps only transitions between two states of procedure candidate, similar to the *mapCore* algorithm, it is more complicated. This complexity arises because the transitions are not just copied. Rather, there exist three possible subtypes of guard transitions (see Table 5.3). The input of the algorithm includes NPDA$_1$, the set of guard transitions T, the stack function Γ_{proc}, which specifies stack symbols for each state of a procedure candidate, and the mapping function Q_{proc}, which maps the original states to the corresponding procedure states. The algorithm maps transitions in two different ways. First, if the pop symbol is equal to ε, this can signify two scenarios: either the transition has not been encountered in any procedure before, necessitating the creation of a new stack symbol, or the transition was a core transition before and represents a shortcut for multiple stack symbols. In either case, the algorithm generates guard transitions for each stack symbol that can appear in the source state. Second, if the pop symbol is a stack

symbol other than ε, the algorithm preserves it and copies the transition with the source and transition states changed according to the function Q_{proc}. Once the mapping of the guard transitions is complete, the resulting automaton is returned.

Algorithm 7: mapGuard

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$,
set of return transitions $T \subseteq (Q \times \Sigma \times \Gamma_\varepsilon \times \Gamma_\varepsilon \times Q)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

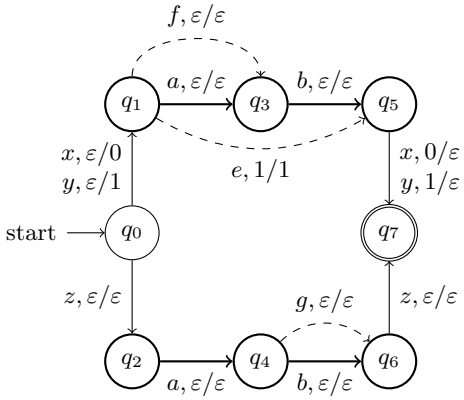
```
1  B ↤ A
2  forall (q, a, α, β, r) ∈ T do
3      q' ↤ Q_proc(q)
4      r' ↤ Q_proc(r)
5      if α = ε then                                   // creating a guard
6          assert(β = ε)
7          forall η ∈ Γ_prov(q) do
8              δ(q', a, η) ↤ δ(q', a, η) ∪ {(r', η)}
9      else                                            // copying a guard
10         assert(β = α)
11         η ↤ α
12         δ(q', a, η) ↤ δ(q', a, η) ∪ {(r', η)}
13 return B
```

$\Gamma_{proc}(q) = 1$	$\alpha = \varepsilon$	$\beta = \alpha$	Is possible	η
✗	✗	✗	no	α
✗	✗	✓	yes	α
✗	✓	✗	no	$\forall\eta \in \Gamma_{proc}(q)$
✗	✓	✓	yes	$\forall\eta \in \Gamma_{proc}(q)$
✓	✗	✗	no	α
✓	✗	✓	no	α
✓	✓	✗	no	$pick(\Gamma_{proc}(q))$
✓	✓	✓	yes	$pick(\Gamma_{proc}(q))$

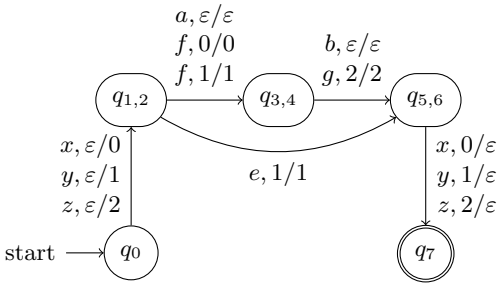
Table 5.3: The table displays all theoretically possible combinations of the source state, pop, and push symbols for guard transitions. It includes all variations of the behavior of the source state, whether it is part of the procedure ($1 < \Gamma_{proc}(q)$) or not ($\Gamma_{proc}(q) = 1$), as well as the specification of the pop symbol (whether it is specified or set to ε) and the push symbols (whether it equals to the pop symbol). The column labeled *Is possible* indicates whether a guard transition with a particular combination is valid according to the semantic of NPDA$_1$. The η column specifies the stack symbol that will be used in the mapped transition as both the pop and push symbol.

As can be seen from Algorithm 7, the condition that distinguishes between the creation of new guard transitions and the copying of the existing one focuses only on the value of the pop symbol α. The reason behind this is shown in Table 5.3. Only three valid combinations of source state behavior, pop and push symbols, highlighted in the table, can be distinguished solely by the value of the pop symbol. The value of the symbol η for the fourth and last rows of the table is determined by the same algorithm operation, as $pick(\Gamma_{proc}(q))$ has the same meaning for $|\Gamma_{proc}(q)| = 1$ as the expression $\forall\eta \in \Gamma_{proc}(q)$.

Valid combinations of the behavior of the source state, pop, and push symbols are highlighted in Table 5.3 and illustrated in Figure 5.4. The other combinations are not possible because they violate the semantics of the NPDA$_1$. The first table row does not denote a valid guard transition (it is a switch transition) and therefore should not be mapped using the *mapGuard* algorithm. The third row of the table describes a situation where the source state is part of some procedure and performs a push without a previous pop. This is not possible since NPDA$_1$ can contain a maximum of one symbol on the stack. The fifth, sixth, and seventh rows represent transitions outside a procedure that use pop or push symbols. This should not be possible, since the stack is empty for states outside procedures.



Automaton $M_{5.4}$ with procedure candidate over bold transitions and states and dashed guard transitions.



The resulting automaton $M_{5.4}$ after the mapping of the guard transitions.

Figure 5.4: This figure illustrates the mapping of guard transitions (dashed transitions) in the automaton $M_{5.4}$ according to *mapGuard* algorithm.

5.2.5 Switch Transitions

Algorithm 8: mapSwitch

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$,
set of return transitions $T \subseteq (Q \times \Sigma \times \Gamma_\varepsilon \times \Gamma_\varepsilon \times Q)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

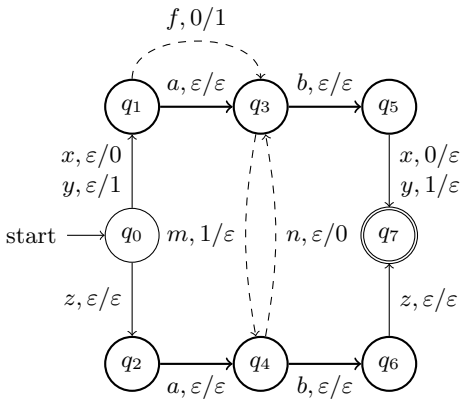
1 $B \leftarrow\!\!\shortmid A$
2 **forall** $(q, a, \alpha, \beta, r) \in T$ **do**
3 $\quad q' \leftarrow\!\!\shortmid Q_{proc}(q)$
4 $\quad r' \leftarrow\!\!\shortmid Q_{proc}(r)$
5 $\quad assert(\alpha = \varepsilon \iff |\Gamma_{proc}(q)| = 1 \wedge \beta = \varepsilon \iff |\Gamma_{proc}(r)| = 1)$
6 $\quad \alpha' \leftarrow\!\!\shortmid \alpha$ **if** $\alpha \neq \varepsilon$ **else** $pick(\Gamma_{proc}(q))$
7 $\quad \beta' \leftarrow\!\!\shortmid \beta$ **if** $\beta \neq \varepsilon$ **else** $pick(\Gamma_{proc}(r))$
8 $\quad \delta(q', a, \alpha') \leftarrow\!\!\shortmid \delta(q', a, \alpha') \cup \{(r', \beta')\}$
9 **return** B

The last type of transitions we need to map are switch transitions, which are either existing transitions with specified different pop and push symbols or transitions between

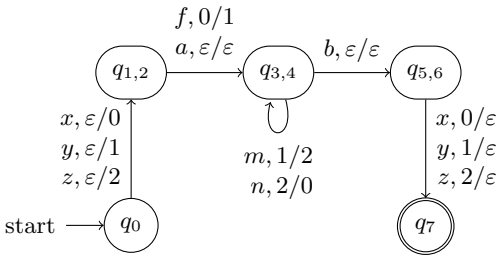
two states where the source state belongs to one transition sequence creating procedure candidate and the destination state belongs to another transition sequence. The algorithm *mapSwitch* takes as input the NPDA$_1$, a set of switch transitions T, the stack function Γ_{proc}, and the mapping function Q_{proc}, which maps the original states to the corresponding procedures states. This algorithm maps four combinations of pop and push symbols. In this mapping, each symbol not specified (equal to ε) is replaced with a new stack symbol provided by the function Γ_{proc}. In the end, the resulting automaton is returned.

$\lvert\Gamma_{proc}(q)\rvert = \lvert\Gamma_{proc}(r)\rvert = 1$	$\alpha = \varepsilon$	$\beta = \varepsilon$	Is possible	α'	β'
✗	✗	✗	yes	α	β
✗	✗	✓	yes	α	$pick(\Gamma_{proc}(r))$
✗	✓	✗	yes	$pick(\Gamma_{proc}(q))$	β
✗	✓	✓	no	$pick(\Gamma_{proc}(q))$	$pick(\Gamma_{proc}(r))$
✓	✗	✗	no	α	β
✓	✗	✓	no	α	$pick(\Gamma_{proc}(r))$
✓	✓	✗	no	$pick(\Gamma_{proc}(q))$	β
✓	✓	✓	yes	$pick(\Gamma_{proc}(q))$	$pick(\Gamma_{proc}(r))$

Table 5.4: Possible combinations of the behavior of the source and target states (whether they both have not been used in any procedure so far, denoted by $\lvert\Gamma_{proc}(q)\rvert = \lvert\Gamma_{proc}(r)\rvert = 1$, or not) and of the pop and push symbols (whether they are specified or set to ε) are considered for switch transitions. The column labeled *Is possible* indicates whether a switch transition with this combination can exist in NPDA$_1$. The α' and β' columns specify the pop and push symbols that will be used on the mapped transition.



Automaton $M_{5.5}$ with procedure candidate over bold transitions and states and dashed switch transitions.



The resulting automaton $M_{5.5}$ after the mapping of the switch transitions.

Figure 5.5: This figure illustrates the mapping of switch transitions (dashed transitions) in the automaton $M_{5.5}$ according to Algorithm 8.

The value of the new pop and push symbols of the mapped switch transition in the *mapSwitch* algorithm depends on the number of stack symbols that can be used within the source and target states, as well as whether the pop and push symbols are specified or set to ε. This results in eight possible combinations, as shown in Table 5.4. However, only four of these combinations are valid switch transitions within the context of an NPDA$_1$.

The first combination represents the valid mapping of an existing switch transition, where both the pop and push symbols are preserved. The second valid combination corresponds to a transition where its source state is a part of an existing procedure (it has to be the source state, not the destination state, otherwise this transition would be invalid in the combination with the pop symbol) and the push symbol is not specified. In this case, the new symbol $pick(\Gamma_{proc}(r))$ is used. The third combination is simply a reverse of the second combination. These three valid switch transitions can be observed in Figure 5.5. The last table row represents the final valid switch transition, where neither the source nor destination states are part of any procedure, and neither the pop nor push symbols are specified. In this scenario, the new symbols $pick(\Gamma_{proc}(q))$ and $pick(\Gamma_{proc}(r))$ are used. Other combinations of the source and target states, pop and push symbols, denote switch transitions or they violate the semantics of the NPDA$_1$. The fourth transition is not a switch transition. It is a core transition and should not be mapped using the *mapSwitch* algorithm. Finally, the fifth, sixth, and seventh combinations represent transitions outside of a procedure that use pop or push symbols. This should not be possible since the stack is empty for states that are not a part of any procedure.

5.2.6 Initial States

The procedure candidate can consist of initial states that specify the set of initial stack symbols that can be placed nondeterministically on the stack of the automaton. It is essential to preserve this behavior while creating procedure states.

The algorithm *mapInitial* takes as input the NPDA$_1$, a procedure candidate P, the stack function Γ_{proc} that specifies stack symbols for each state of a procedure candidate, and the mapping function Q_{proc} that maps original states to the corresponding procedure states. If the original initial state i is mapped to the procedure state s, then the state s is marked as initial, and the function $\lambda(s)$ is extended by the set of initial stack symbols $\lambda(i)$ if it is not equal to $\{\varepsilon\}$. Otherwise, the set of stack symbols $\Gamma_{proc}(i)$ is used. The resulting automaton with the updated function λ is returned.

Algorithm 9: mapInitial

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta, I_{in}, \lambda_{in}, F, \phi)$,
procedure candidate $P = (V, E)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1 B ↤ A
2 forall q ∈ Q(P) ∩ I do
3     q′ ↤ Q_proc(q)
4     I ↤ I ∪ {q′}
5     Ψ ↤ λ_in(q) if λ_in(q) ≠ {ε} else Γ_proc(q)
6     λ(q′) ↤ λ(q′) ∪ Ψ
7 return B
```

5.2.7 Final States

In addition to mapping initial states, it in crucial to also map the behavior of final states because each final state can specify the set of stack symbols that should be on the top of the stack when the automaton accepts the input.

The algorithm *mapFinal* takes as input the NPDA$_1$, a procedure candidate P, the stack function Γ_{proc} that specifies stack symbols for each state of a procedure candidate, and the mapping function Q_{proc} that maps the original states to the corresponding procedure states. The algorithm maps the final states in such a way that if the final state p is mapped to the procedure state s, then s becomes a final state. The function $\phi(s)$ is extended by the set of stack symbols $\phi(p)$ if it is not equal to $\{\varepsilon\}$. Otherwise, the set of stack symbols $\Gamma_{proc}(p)$ is used. The resulting automaton with the updated function ϕ is returned.

Algorithm 10: mapFinal

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F_{in}, \phi_{in})$,
procedure candidate $P = (V, E)$,
stack function $\Gamma_{proc} : Q \to 2^{\Gamma_\varepsilon}$, and state function $Q_{proc} : Q(P) \to Q$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1 B ↤ A
2 forall q ∈ Q(P) ∩ F do
3     q' ↤ Q_proc(q)
4     F ↤ F ∪ {q'}
5     Ψ ↤ φ_in(q) if φ_in(q) ≠ {ε} else Γ_proc(q)
6     φ(q') ↤ φ(q') ∪ Ψ
7 return B
```

5.3 Reduction of Stack Alphabet

After the Procedure mapping is complete and there are no more procedure candidates with positive gains left, the stack alphabet often contains many symbols that cannot coexist in the same state. This occurs because the stack symbols typically do not meet: either they exist in isolated procedures or each belongs to a different procedure branch that does not intersect with each other after the branching. In such scenarios, the stack alphabet can be simplified by representing symbols, that cannot coexist, with the same value.

By applying this approach, the size of the resulting stack alphabet will be equal to the size of the maximal set of symbols that can coexist at the same state. Additionally, this approach can indirectly simplify the automaton by transforming switch transitions, which often occur at the branching of a procedure, to guard transitions. This transformation of transitions has a potential to further simplify the automaton by replacing multiple guard transitions with a single core transition.

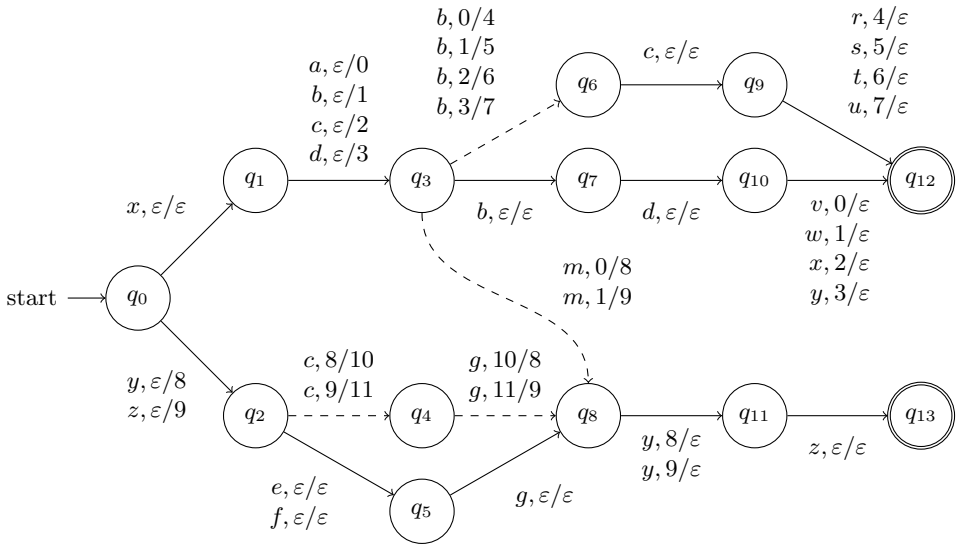
To identify stack symbols that can coexist in the same state, we establish an equivalence relation on stack symbols called *meet*. This relation allows us to create equivalence classes, enabling us to rename the stack symbols and obtain the smallest possible stack alphabet. The maximum number of symbols after the renaming will correspond to the size of the largest equivalence class.

Definition 5.1. *Let $A = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$ be an NPDA$_1$. The binary relation **meet** on Γ is defined as $\wedge_\Gamma \subseteq \Gamma \times \Gamma$, such that $(\alpha, \beta) \in \wedge_\Gamma \iff \exists q \in Q : \alpha \in \gamma(q) \wedge \beta \in \gamma(q)$.*

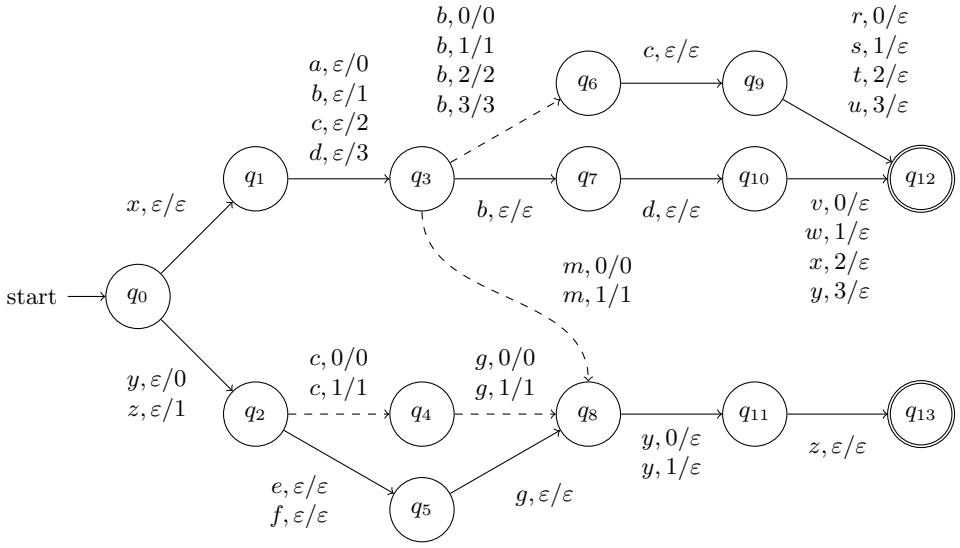
Theorem 5.1. *The relation $\wedge_\Gamma$ is an equivalence relation.*

Proof. The proof of the equivalence (reflexivity, symmetry, and transitivity) of the relation $\wedge_\Gamma$ is trivial and can be demonstrated using the properties of logical conjunction and set membership. Therefore, it is omitted and left to the reader as an exercise. □

Since the relation $\wedge_\Gamma$ is an equivalence relation, it partitions the set Γ into equivalence classes. We denote the equivalence class of the symbol α as $[\alpha]_{\wedge\Gamma} = \{\beta \in \Gamma \mid (\alpha, \beta) \in \wedge_\Gamma\}$. The set of all equivalence classes is denoted as $[\Gamma]_{\wedge\Gamma} = \{[\alpha]_{\wedge\Gamma} \mid \alpha \in \Gamma\}$. These classes are utilized by the algorithm *reduceStack* to rename the stack symbols and achieve the smallest possible stack alphabet size.



Automaton $M_{5.6}$ before the reduction of the stack alphabet, with a stack alphabet size of 12. Dashed transitions suffer the most from multiple stack symbols.



Automaton $M_{5.6}$ after reducing the stack alphabet by the renaming of the stack symbols, the dashed transitions have transformed into guard transitions.

Figure 5.6: This figure shows the automaton $M_{5.6}$ before and after the reduction of the stack alphabet. The reduction has transformed switch transitions into guard transitions.

The algorithm *reduceStack* takes as input NPDA_1 and returns the automaton with the stack that is reduced to the minimal possible size. First, it constructs the equivalence relation $\wedge_\Gamma$. The algorithm then creates the function $rename : \Gamma_\varepsilon \to \mathbb{N}_0 \cup \{\varepsilon\}$, which assigns a new name to each stack symbol in every equivalence class, except for the symbol ε. The symbol ε is never renamed. The construction of the new reduced stack alphabet and the renaming of the stack symbols that are used within functions λ_{in} and ϕ_{in} follow. The final step involves renaming all push and pop symbols in the transition function δ_{in}. In the end, the resulting automaton with the reduced size of the stack alphabet is returned. The potential for further reduction in the number of automaton's transitions lies in the compression of multiple guard transitions, that arised from the renaming of the stack symbols, into a single core transition.

Algorithm 11: reduceStack

Input: NPDA_1 $A = (Q, \Sigma, \Gamma_{in}, \delta_{in}, I, \lambda_{in}, F, \phi_{in})$
Output: NPDA_1 $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

$B \leftarrow (Q, \Sigma, \emptyset, \emptyset, I, \emptyset, F, \emptyset)$ `// ∅ denotes an empty relation or function`

`// Constructing rename function for stack symbols using the relation` $\wedge_\Gamma$
$\wedge_\Gamma \leftarrow \{(\alpha, \beta) \in \Gamma_{in} \times \Gamma_{in} \mid \alpha \neq \varepsilon \wedge \beta \neq \varepsilon \wedge \exists q \in Q : \alpha \in \gamma(q) \wedge \beta \in \gamma(q)\}$
forall $[\alpha]_{\wedge_\Gamma} \in [\Gamma]_{\wedge_\Gamma}$ **do**
    $\eta \leftarrow 0$
    forall $\beta \in [\alpha]_{\wedge_\Gamma}$ **do**
        $rename(\beta) \leftarrow \eta$
        $\eta \leftarrow \eta + 1$

`// Epsilon has to be handled separately`
$\wedge_\Gamma \leftarrow \wedge_\Gamma \cup \{(\varepsilon, \varepsilon)\}$
$[\Gamma]_{\wedge_\Gamma} \leftarrow [\Gamma]_{\wedge_\Gamma} \cup \{\{\varepsilon\}\}$
$rename(\varepsilon) \leftarrow \varepsilon$

`// Renaming`
$\Gamma_\varepsilon \leftarrow \{rename(\alpha) \mid \alpha \in \Gamma_{in}\}$
forall $q \in I$ **do**
    $\lambda(q) \leftarrow \{rename(\alpha) \mid \alpha \in \lambda_{in}(q)\}$
forall $q \in F$ **do**
    $\phi(q) \leftarrow \{rename(\alpha) \mid \alpha \in \phi_{in}(q)\}$
forall $(q, a, \alpha) \in dom(\delta_{in})$ **do**
    $\delta(q, a, rename(\alpha)) \leftarrow \{(r, rename(\beta)) \mid (r, \beta) \in \delta_{in}(q, a, \alpha)\}$
return B

5.4 Reduction of Guard Transitions

The reduction of the stack alphabet not only minimizes the number of stack symbols used in the automaton, but also, by this reduction, some transitions are transformed from switch transitions to guard transitions. To further simplify the automaton, the set of newly created guard transitions can be replaced with a single core transition. This replacement can be made if the guard transitions have the same source and destination states, the same alphabet symbol, and utilize all stack symbols that can occur within the source state. This transformation is possible because the transition from the source state to the target state over an alphabet symbol would occur regardless of the content of the stack.

The algorithm *reduceGuard* takes as input the NPDA$_1$ and returns the automaton with guard transitions replaced by one core transition when possible. The algorithm first computes the set of guard transitions that can be replaced with a core transition. These guard transitions are gathered according to the source state, the alphabet symbol, and the destination state. It is important to note that a set of guard transitions between the source and destination states for the same alphabet symbol can be replaced only if there is a guard transition for each stack symbol that can occur on top of the stack in the source state. Otherwise, the replacement of the guard transitions by a core transition is not possible because it would allow the remaining stack symbols, which have not been used in guard transitions, to access the part of the procedure where they should not be able to go, thus overapproximating the language of the automaton. After this step, the new transition function is created by replacing guard transitions, from the previously computed set *transform*, with single core transitions. Finally, the resulting automaton with a reduced number of guard transitions is returned.

Algorithm 12: reduceGuard

Input: NPDA$_1$ $A = (Q, \Sigma, \Gamma_\varepsilon, \delta_{in}, I, \lambda, F, \phi)$
Output: NPDA$_1$ $B = (Q, \Sigma, \Gamma_\varepsilon, \delta, I, \lambda, F, \phi)$, such that $L(A) \equiv L(B)$

```
1  B ↤ (Q, Σ, Γ_ε, ∅, I, λ, F, φ)                          // ∅ denotes an empty function

   // Guard transitions that are able to be transformed into core transition
2  transform ↤ {(q, a, r) | ∀α ∈ γ(q) : (r, α) ∈ δ_in(q, a, α), where q, r ∈ Q, a ∈ Σ}

   // Creating transitions
3  forall (q, a, α) ∈ dom(δ_in) do
4      forall (r, β) ∈ δ_in(q, a, α) do
5          if (q, a, r) ∈ transform then
6              δ(q, a, ε) ↤ δ(q, a, ε) ∪ {(r, ε)}            // make core transition
7          else
8              δ(q, a, α) ↤ δ(q, a, α) ∪ {(r, β)}            // keep guard transition
9  return B
```

The benefits of reducing guard transitions can be seen in Figure 5.7. Not only does the replacement of guard transitions by core transitions reduce the number of transitions, but it also allows for the existence of new language equivalent states that can be merged into one (the utilization of this merge is not part of our algorithm, but can be added in future work). For example, the states q_6 and q_7, or the states q_4 and q_5, can be merged into one state in the automaton $M_{5.7}$.

Figure 5.7 also demonstrates both the potential benefit and the risk that arises from replacing guard transitions that do not cover all possible stack symbols with one core transition. One such scenario can be illustrated by a set of transitions between states q_3 and q_8 over the alphabet symbol m. Here, these transitions cannot be replaced by a core transition due to the absence of guard transitions for the stack symbols 2 and 3. However, if we were to replace the transitions regardless (obtaining a more minimal automaton), the stack symbols 2 and 3 would gain access to state q_8, where they would normally not be allowed. Although in this specific case, this might not lead to any issues, since it is impossible to proceed further from state q_8 using the stack symbols 2 or 3, this approach cannot be used in general because it can overapproximate the language of the automaton.

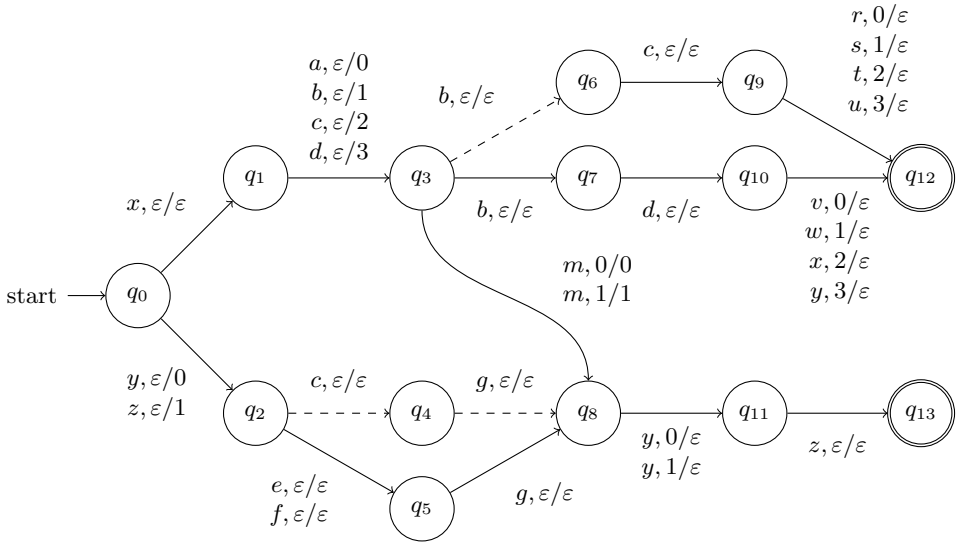


Figure 5.7: The automaton after applying *reduceGuard* algorithm on the resulting automaton $M_{5.6}$ from the algorithm *reduceStack*. The dashed transition indicates the transitions that were transformed by the algorithm from guard transitions to core transitions.

Consider a scenario showed by Figure 5.8 that will alter the language of the automaton. The guard transitions between states q_6 and q_5 can be correctly replaced by a single core transition. However, replacing the set of guard transitions between states q_3 and q_4, which lacks a guard transition for the stack symbol 2, would result in an overapproximation of the automaton language. Because this change introduces a new possible path for stack symbol 2 over states q_1, q_3, q_4, q_6, q_5, to the accepting state q_7. This path should be accessible only with the symbols 0 or 1 on top of the stack.

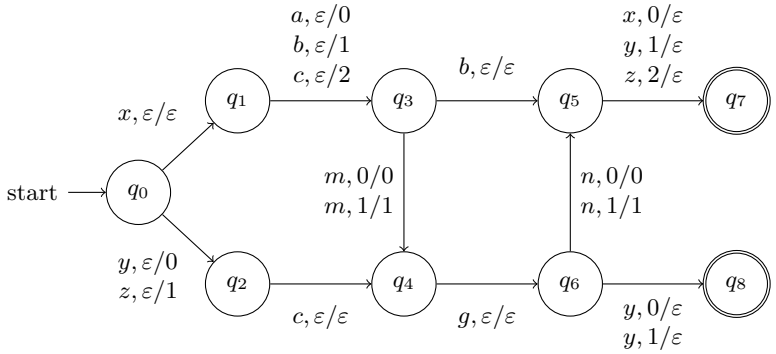


Figure 5.8: This example illustrates a situation where the replacement of guard transitions between states q_6 and q_5 with one core transition is followed by the dangerous replacement of guard transitions between states q_3 and q_4, which alters the language of the automaton.

Although the technique of unsafe replacement of guard transitions by one core transition cannot be utilized in the current version of our algorithm due to the risk of overapproximating the automaton's language, it represents an interesting approach that, with additional conditions, could be employed in future work for a better reduction of automata.

Chapter 6

Experimental Results

In this chapter, we will present a comparison between automata reduced only by state merging and transition pruning, and these reductions enhanced by our Procedure mapping algorithm. This new reduction approach has been extensively benchmarked on 52,474 automata of various types to show its versatility. The automata were obtained from Abstract Regular Model Checking [7], the library of regular expressions RegexLib [33], parametrized regular expressions [21], the network intrusion detection system Snort [37, 44], string constraints from [1] and SyGuS-qgen [35] (collected in the SMT-LIB benchmark [34]), regular expression membership [39], the Z3-noodler tool for string solving [24, 36], and decisions of WS1S formulae that were used for the evaluation of the Gaston tool [18].

The results were computed on a machine with an AMD Ryzen 7 3800XT 8-Core processor and 32 GB of RAM. First, automata were reduced by the RABIT/Reduce tool (version 2.5) implemented in Java, which performs minimization using state merging and transition pruning with an advanced variant of simulation [11] with a look-ahead of 12. Our Procedure mapping algorithm, implemented in Python 3.11.6, was then further applied to the resulting automata. To expedite the calculation of the procedure candidate gain for each vertex in the superproduct, the algorithm used a look-ahead of 10, meaning that the approximate gain of the procedure candidate starting from vertex v was calculated based only on the successors of v that were at most 10 steps away.

The experiments witnessed a significant reduction compared to the simple use of the RABIT/Reduce tool, which reduced, on average, the number of states by 44.1% and the number of transitions by 46.2%. Furthermore, by applying the Procedure mapping algorithm, we were able to achieve, on average, an additional reduction of 29.5% in the number of states and 30.9% in the number of transitions. The most significant reduction, obtained on automaton generated during the computation of Z3-noodler tool, was 64.9% in states and 67.3% in transitions. The more detailed results follow in the next sections.

As we mentioned earlier in Section 4.5, the number of states in the resulting automaton is not the only criterion for the reduction in the number of states. Each automaton can be trivially reduced to one state by transferring all the logic onto the stack, leading to an increase in the stack alphabet by the number of states in the original automaton. Therefore, the reduction in the number of states has to be calculated from the sum of the number of states and the number of stack symbols (before applying *reduceStack* algorithm) in the resulting automaton. This behavior does not apply to the number of transitions. The reduction in the number of transitions is determined only by the number of transitions itself.

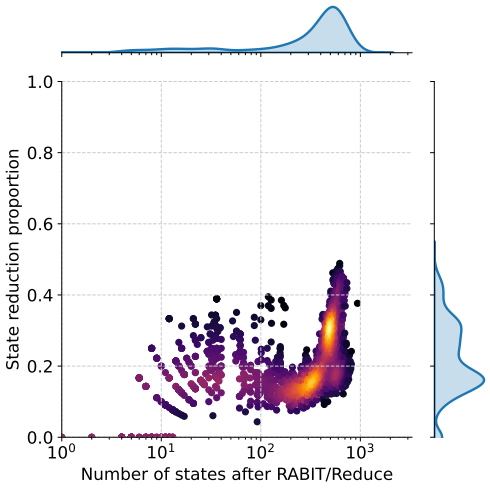
6.1 Abstract Regular Model Checking

The benchmarking set contained automata that were created during Abstract Regular Model Checking [7], consisting of 2,604 automata with an average of 863 states and 2,999 transitions, and a maximum of 2,591 states and 12,971 transitions. The automata were first reduced by the RABIT/Reduce tool, utilizing state merging and transition pruning, and then by our algorithm.

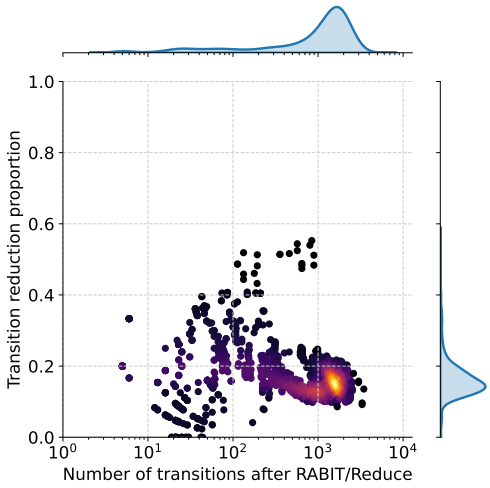
×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	863	2,999	386	1,207	164+124	(74.6%)	1,021	(84.5%)	9
Best Q	1,518	6,397	620	2,186	188+129	(51.1%)	1,744	(79.8%)	6
Best δ	459	4,445	176	843	50+62	(63.6%)	377	(44.7%)	6

Table 6.1: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.

From Table 6.1, it can be seen that the standalone usage of the RABIT/Reduce tool reduced, on average, the number of states by 55.3% and the number of transitions by 59.8% compared to the original automata. Such automata were the further reduced by our algorithm, resulting in an additional 25.4% reduction in the number of states and 15.4% reduction in the number of transitions. Reductions for all automata from Abstract Regular Model Checking are illustrated in Figure 6.1.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.1: The reductions for automata from Abstract Regular Model Checking, after applying the Procedure mapping algorithm to the results from the RABIT/Reduce tool, showed an average reduction of 25.4% in states and 15.4% in transitions.

6.2 Regular Expressions

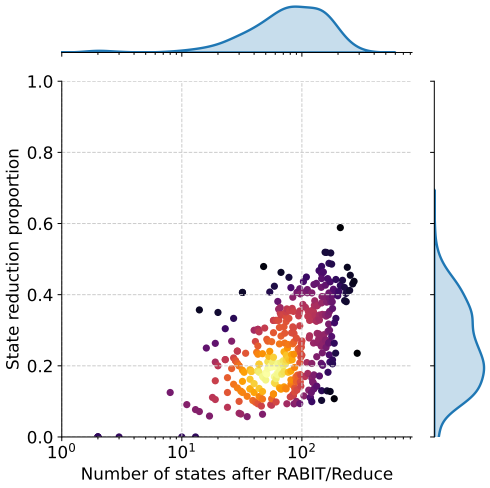
In this section we tested the Procedure mapping algorithm on automata created from two sets of regular expressions from real-world scenarios. The first set, used to validate email addresses, was obtained from the RegexLib library [33], which contains more than 4,000 regular expression patterns for tasks such as email validation, phone number detection, URL detection, etc. The second set consists of diverse regular expressions collected from questions across Stack Overflow.[1]

6.2.1 Email Validation

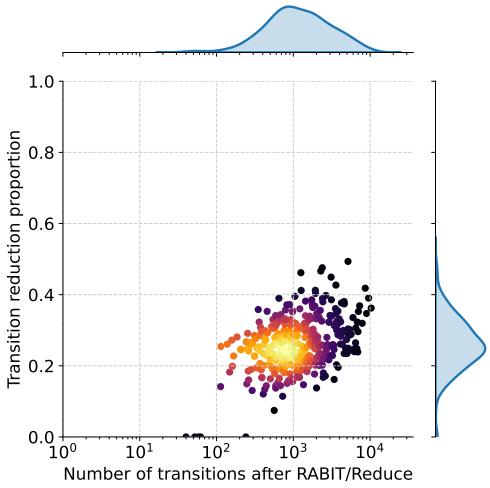
We selected the top 75 regular expressions for email validation from the RegexLib library and created 362 automata following the approach outlined in [14] and [41]. These automata contained average of 115 states and 1,962 transitions, with a maximum of 451 states and 24,456 transitions. Initially, we applied the tool RABIT/Reduce to reduce the automata, followed by our Procedure mapping algorithm.

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	115	1,962	93	1,759	48+18	(71.0%)	1,256	(71.4%)	5
Best Q	324	333	324	333	92+34	(38.9%)	123	(36.9%)	10
Best δ	263	5,793	250	5,149	107+40	(58.8%)	2,608	(50.7%)	8

Table 6.2: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.2: The reductions for automata created from regular expressions validating email addresses, after applying the Procedure mapping algorithm to results from the reduction tool RABIT/Reduce, showed an average reduction of 29% in state and 28.6% in transition.

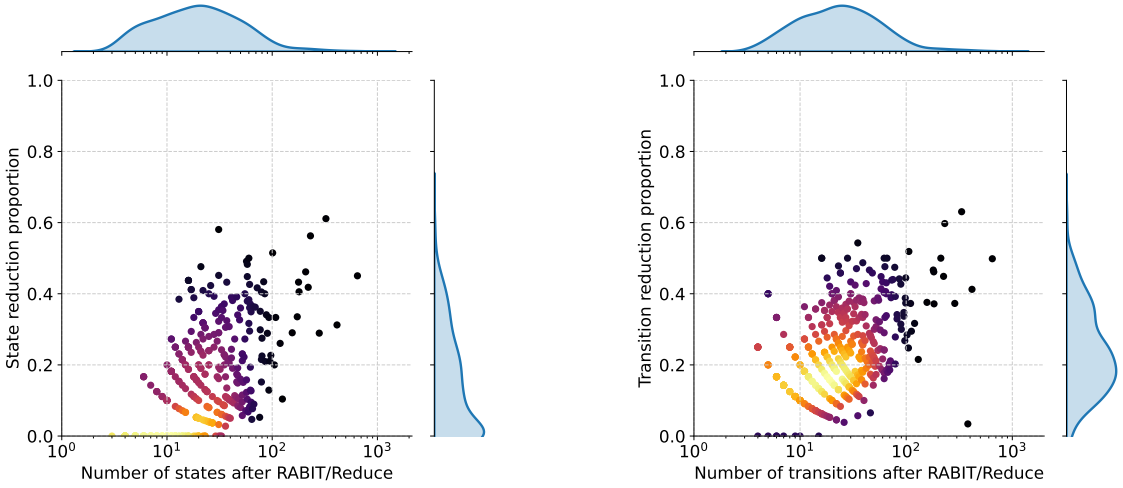
[1] https://stackoverflow.com

The average reduction results and the best reduction results are summarized in Table 6.2. The RABIT/Reduce tool reduced the average automaton by 19.1% in the number of states and 10.3% in the number of transitions. Additionally, by applying the Procedure mapping algorithm, the automata were further reduced by 29% in the number of states and 28.6% in the number of transitions. The reduction results for all generated automata are illustrated in Figure 6.2.

The best reduction occurred in the automaton with 324 states and 333 transitions. Surprisingly, the reduction tool RABIT/Reduce could not perform any reduction on this automaton. However, after applying the Procedure mapping algorithm, the automaton was reduced to only 92 states with 34 stack symbols, representing a reduction of 61.1% in states, and the number of transitions was reduced by 63.1% to 123 transitions.

6.2.2 Stack Overflow

The second set contained 430 automata created from various regular expressions collected from answers and questions on Stack Overflow. The automata contained on average 32 states and 37 transitions, with a maximum of 646 states and 648 transitions. They were initially reduced by RABIT/Reduce, and then further by our Procedure mapping algorithm.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.

The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.3: The reductions for automata created from expressions collected on Stack Overflow, after applying the Procedure mapping algorithm to the results from the reduction tool RABIT/Reduce, showed an average reduction of 28.1% in states and 30.6% in transitions.

The RABIT/Reduce tool did not achieve any noticeable reduction. On average, it was unable to reduce the number of states and only reduced the number of transitions by 2.7%. In contrast, although the automata were small, the Procedure mapping algorithm reduced the number of states by 28.1% and the number of transitions by 30.6% on average. The results of the reduction for all automata are illustrated in Figure 6.3.

The reduction statistics can be seen in Table 6.3. The most notable reduction was achieved on the automaton with 324 states and 333 transitions. The RABIT/Reduce tool was unable to perform any reductions on this automaton. However, applying the Procedure

mapping algorithm resulted in the automaton having only 92 states with 34 stack symbols, representing a remarkable reduction of 61.1% in states, and the number of transitions was reduced to 123, indicating a reduction of 63.1%.

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	32	37	32	36	17+6	(71.9%)	25	(69.4%)	2
Best Q nad δ	324	333	324	333	92+34	(38.9%)	123	(36.9%)	10

Table 6.3: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.

6.3 Parametric Regular Expressions

In this section, we evaluated the reduction of 3,656 automata with an average of 207 states and 2,584 transitions, and with a maximum of 504 states and 12,144 transitions. The automata were created from four parametric regular expressions, with parameters $e, i, n \in \mathbb{N}$, where $i \leq n$ and e grows exponentially, and $\emptyset \neq \phi \subseteq \Sigma$. These expressions were obtained from [21], particularly:

1. `([0-1]{`$i-1$`}0[0-1]{`$n-1$`}0[0-1]{`$n-i$`}`ϕ`) |`
   `([0-1]{`$i-1$`}1[0-1]{`$n-1$`}1[0-1]{`$n-i$`}`ϕ`)`
2. `[a-c]*([a-c]{`e`})+`ϕ
3. `[a-c]+`ϕ`d[a-c]{`e`}+`
4. intersection of `[a-c]*a[a-c]{`$i+1$`}` and `[a-c]*b[a-c]{`i`}`

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	207	2,584	93	1,332	42+6	(51.6%)	694	(52.1%)	3
Best Q	359	594	182	417	61+7	(37.4%)	187	(44.8%)	4
Best δ	182	7,550	63	3,716	20+9	(46.0%)	1,534	(41.3%)	4

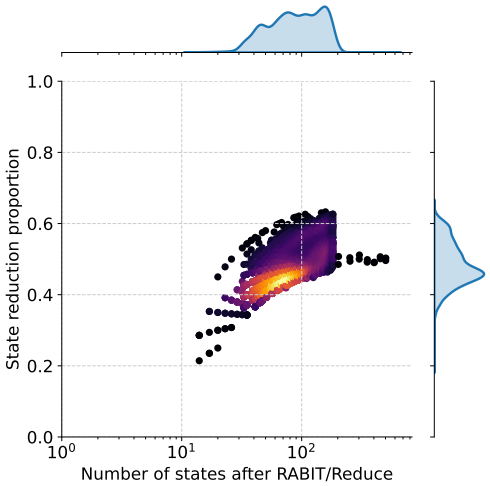
Table 6.4: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.

The results of the reduction achieved by the Procedure mapping algorithm on automata, created from parametric regular expressions, are summarized in Figure 6.4. The simple usage of the RABIT/Reduce tool resulted, on average, in 55.1% reduction in states and 48.5% reduction in transitions. Further application of our algorithm resulted in an additional 48.4% reduction in states and 47.9% reduction in transitions.

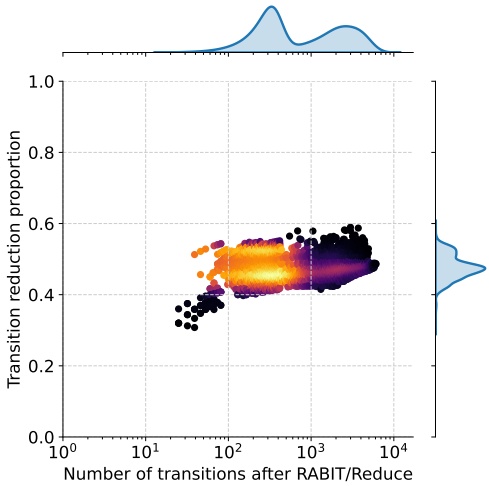
The most significant reduction in terms of the number of states was achieved on the automaton generated from the first family of parametric regular expressions. The further application of the Procedure mapping algorithm to this automaton resulted in the automaton with only 61 states and 7 stack symbols, representing a 62.6% reduction in the number of states, and 187 transitions, reflecting a 55.2% reduction in transitions.

The best reduction of automaton transitions was achieved on the automaton generated from the second family of parametric regular expressions. Applying the Procedure mapping

algorithm after RABIT/Reduce resulted in the automaton containing 20 states and 9 stack symbols, indicating a 53.9% reduction in states, and 1,534 transitions, representing a 58.7% reduction in transitions.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.4: The reductions, for automata created from parametrized regular expressions, after applying the Procedure mapping algorithm to the results from the RABIT/Reduce tool. On average, we achieved a reduction of 48.4% in states and 47.9% in transitions.

6.4 Snort

The Snort tool is a network intrusion detection system (NIDS) that uses a set of rules to detect malicious traffic [37]. These rules consist of various attributes, including protocol (TCP, UDP, ICMP), source and destination addresses, source and destination ports, regular expressions for matching traffic payloads (PCREs), etc. To cope with high-speed networks, the traffic can be pre-filtered based on the union of regular expressions. This pre-filtering can significantly reduce the amount of traffic that the NIDS needs to handle, often by two or three orders of magnitude [28].

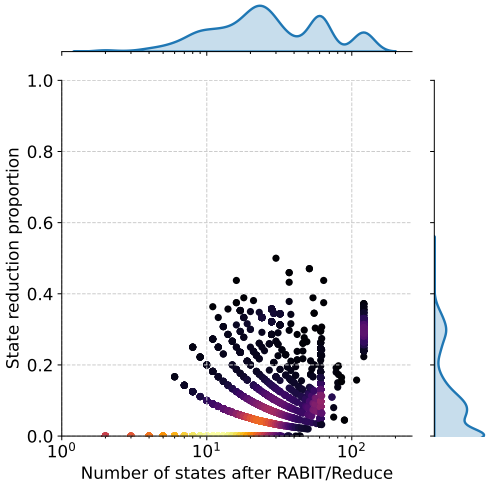
In our experiments, we utilized regular expressions from seven families of Snort rules and constructed automata from them using the Netbench tool [31]. In this section, we will first evaluate the reduction potential of our approach compared to the RABIT/Reduce tool on individual automata created from individual PCREs. Following this, we will proceed to test the reduction on seven automata, where each automaton represents a union of regular expressions from one family of Snort rules.

6.4.1 Individual Rules

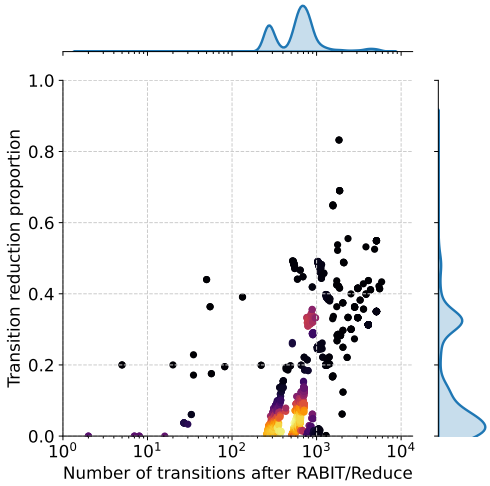
We created 3,616 automata from seven families of Snort rules [44] specifically from the rules `p2p`, `worm`, `shellcode`, `mysql`, `chat`, `specific-threats`, and `telnet`. The automata for single regular expressions contained 41 states and 1,162 transitions on average, and a maximum of 135 states and 34,968 transitions. These automata were reduced by the

RABIT/Reduce tool and then by our Procedure mapping algorithm. The average and the best reduction can be seen in Table 6.5.

The use of the RABIT/Reduce tool reduced, on average, the number of states by 9.8% and the number of transitions by 33.1%. The further application of our Procedure mapping algorithm resulted in an additional reduction of the number of states by 18.9% and the number of transitions by 21.8%. The results of all reductions are shown in Figure 6.5.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.5: The reductions, for automata constructed from PCREs from the Snort rules, after applying the Procedure mapping algorithm to the results from the RABIT/Reduce tool. On average, we achieved a reduction of 18.9% in states and 21.8% in transitions.

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	41	1,162	37	777	20+10	(81.1%)	608	(78.2%)	3
Best Q	31	796	30	285	11+4	(50.0%)	269	(94.4%)	4
Best δ	48	3,366	42	1,842	19+14	(78.6%)	309	(16.8%)	7

Table 6.5: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.

The best state reduction was achieved on the automaton constructed from the regular expression `.*\x00{2}\x03\xe7\x00{7}\x65\x00{8}\x0d\x05\x00{7}` from the set of rules `snort-default`. The RABIT/Reduce tool reduced the automaton by 3.2% in the number of states and by 64.2% in the number of transitions. On the other hand, further application of the Procedure mapping algorithm reduced the automaton by an additional 50% in the number of states but only 5.6% in the number of transitions.

The most significant reduction of transitions was achieved on the automaton constructed from the regular expression `\x7BIP\x7D[^\x7D\r\n]\x7BOS\x7D[^\x7D\r\n]\x7BUptime \x7D[^\x7D\r\n]\x7BTrojan\x7D[^\x7D\r\n]\x7BPSW\x7D[^\x7D\r\n]\x7BPort\x7D[`

`^\x7D\r\n]\x7BUser\x7D` from the set of rules `spyware-put`. The RABIT/Reduce tool reduced the automaton by 12.5% in states and by 45.3% in transitions. The further application of the Procedure mapping algorithm reduced the automaton by an additional 21.4% in states and a surprising 83.2% in transitions.

6.4.2 Union of Rules

The reduction potential of our method was also tested on seven larger automata, each created as a union of regular expressions from one of the seven families of Snort rules. The automata consisted of between 33 and 829 states, with between 1,090 and 57,292 transitions. The results of the reduction for each set of rules, with the two most significant reductions highlighted, can be seen in Table 6.6.

Snort rules	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
p2p	33	1,090	32	1,084	25+6	(96.9%)	570	(52.6%)	2
worm	50	3,880	34	290	24+8	(94.1%)	284	(97.9%)	2
shellcode	162	3,328	56	579	48+2	(89.3%)	486	(83.9%)	2
mysql	235	30,052	91	14,430	45+18	(69.2%)	7,142	(49.5%)	5
chat	408	23,937	113	1,367	71+25	(85.0%)	1,058	(77.4%)	3
specific-threats	459	57,292	236	31,935	99+32	(55.5%)	12,680	(39.7%)	6
telnet	829	7,070	309	2,898	155+82	(76.7%)	2,164	(74.7%)	4

Table 6.6: Results of applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$) algorithms to seven sets of Snort rules. Γ_{Map} represents the stack alphabet, while Γ'_{Map} denotes the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages provided refer to the results obtained from RABIT/Reduce.

It can be seen that the best reduction was achieved on the automaton created from the `specific-threats` rule set. The RABIT/Reduce tool reduced the automaton by 48.6% in the number of states and by 44.3% in the number of transitions. Further application of the Procedure mapping algorithm reduced the number of automaton states by another 44.5% and the number of transitions by another 60.3%.

6.5 String Constraints and Membership

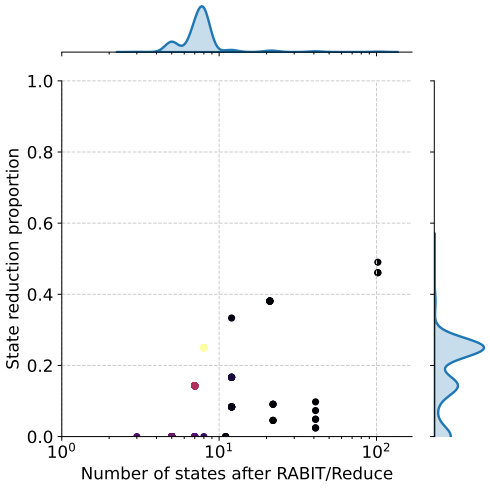
This section presents the reduction results on automata collected from two sources. The first set comprises 452 automata obtained from string constraints found in Norn [1] and SyGuS-qgen [35], both sourced from SMT-LIB benchmarks [34]. The second set contains automata designed to test membership in regular expressions extended with intersection or complementation. These automata are derived from four particularly challenging handwritten problems described in [39]: 1) problems related to dates, 2) problems concerning passwords, 3) problems where Boolean operations interact with concatenation and iteration, and 4) problems with exponential determinization. On average, the automata contained 13 states and 59 transitions, with a maximum of 205 states and 1,457 transitions.

As shown in Table 6.7, standalone usage of the RABIT/Reduce tool achieved an average reduction of 38.5% in states and 57.6% in transitions. Further use of our Procedure mapping algorithm led to an additional reduction of 25% in states and 32% in transitions. Despite the small size of the automata on the input of our algorithm, it is evident that the reduction achieved by the Procedure mapping was significant. The results of the reduction for all automata can be seen in Figure 6.6.

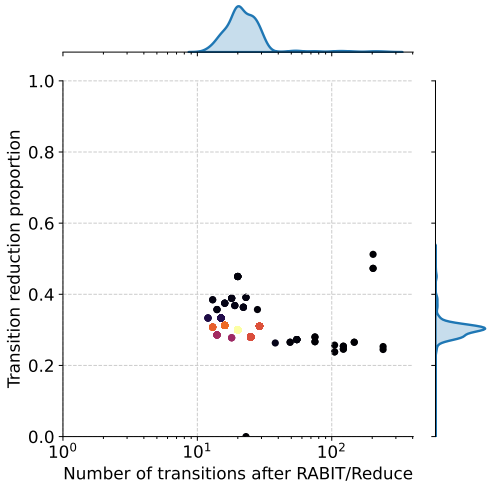
The best reduction result was achieved on the automaton that describes problems with exponential determinization. The RABIT/Reduce tool reduced the automaton by 50.2% in the number of states and by 51.1% in the number of transitions. The subsequent application of the Procedure mapping algorithm resulted in an additional reduction of 49% in the number of states and 51.2% in the number of transitions.

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map} + \Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	13	59	8	25	4+2	(75.0%)	17	(68.0%)	2
Best Q and δ	205	415	102	203	47+5	(51.0%)	99	(48.8%)	3

Table 6.7: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map} + \Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.



The reductions in the number of states ($Q_{Map} + \Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.6: The reductions, for automata containing string constraints and problems of membership in regular expressions, after applying the Procedure mapping algorithm to results from the RABIT/Reduce tool. On average, we achieved a reduction of 25% in the number of states and 32% in the number of transitions transitions.

6.6 Z3-noodler

In this section, we evaluated the Procedure mapping algorithm on a set of automata that were obtained from the Z3-noodler [24] string solver, together with the intended automata operations that are applied within the tool. The automata were generated during the solution of the constraints from the SMT Competition [36]. Our test set contains 819 automata with 63 states and 991 transitions on average, and with a maximum of 553 states and 104,546 transitions. These automata were first reduced by the RABIT/Reduce tool, followed by the Procedure mapping algorithm. The average and best results of the reduction can be seen in Table 6.8.

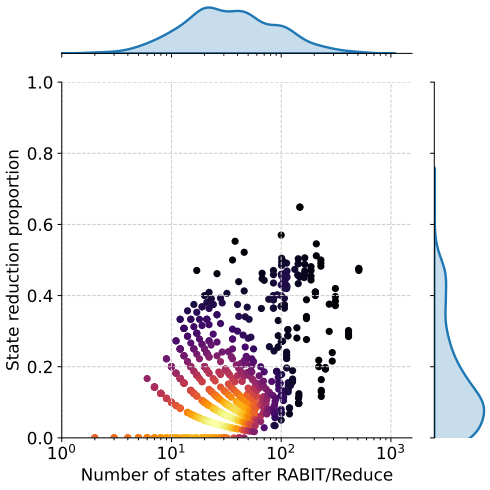
On average, the RABIT/Reduce tool reduced automata by 14.3% in the number of states and by 54% in the number of transitions. The further application of the Procedure mapping algorithm resulted in an additional reduction of 25.9% in the number of states and 39.5% in the number of transitions. Figure 6.7 illustrates all the reduction results.

The best reduction in the number of states and transitions can be seen in the automaton generated by Z3-noodler during complementation. This automaton contained 148 states and an extreme 30,652 transitions. The RABIT/Reduce tool was not able to reduce the number of states. However, using transition pruning, it reduced transitions by a surprising 92.4%. The further application of the Procedure mapping algorithm finally reduced states by 64.9% and transitions by another 67.3%.

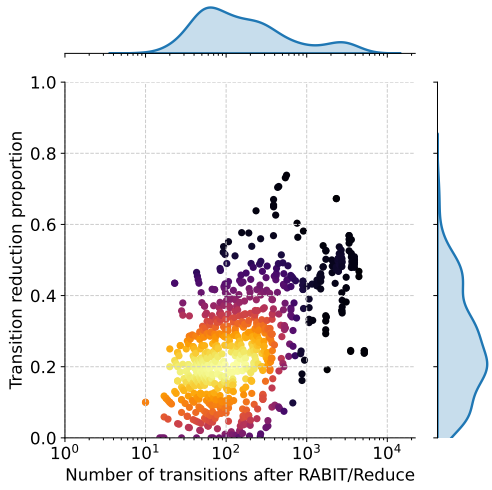
The most significant reduction of transitions was also achieved on the automaton generated during complementation. The automaton was reduced, using the RABIT/Reduce tool, by 10.4% of states and only 1.9% of transitions. On the other hand, the further utilization of the Procedure mapping algorithm reduced the automaton by 14.7% in states and by an astonishing 73.8% in transitions.

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map}+\Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	63	991	54	456	29+11	(74.1%)	276	(60.5%)	3
Best Q	148	30,652	148	2,332	43+9	(35.1%)	763	(32.7%)	5
Best δ	106	573	95	562	50+31	(85.3%)	147	(26.2%)	7

Table 6.8: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map}+\Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.



The reductions in the number of states ($Q_{Map}+\Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.7: The reductions, for automata created by the Z3-noodler tool, after applying the Procedure mapping algorithm to the results from the RABIT/Reduce tool. On average, we achieved a reduction of 25.9% in states and 39.5% in transitions.

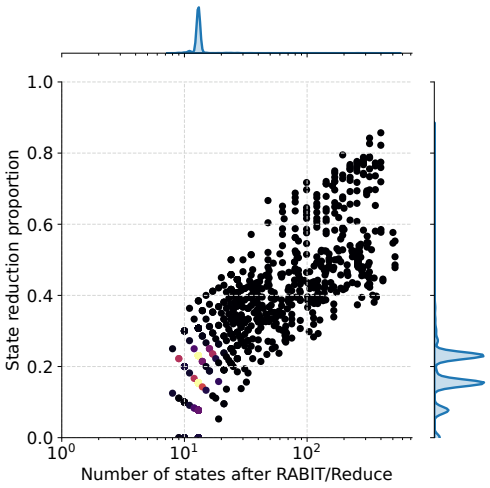
6.7 WS1S

Last but not least, we compared the reduction achieved by our Procedure mapping algorithm against that of the RABIT/Reduce tool on five families of automata created from symbolic automata generated during the computation of the tool that was a part of Pavel Bednář's Master's thesis [6], to decide WS1S formulae. These formulae were previously used during the evaluation of the Gaston tool [18], particularly:

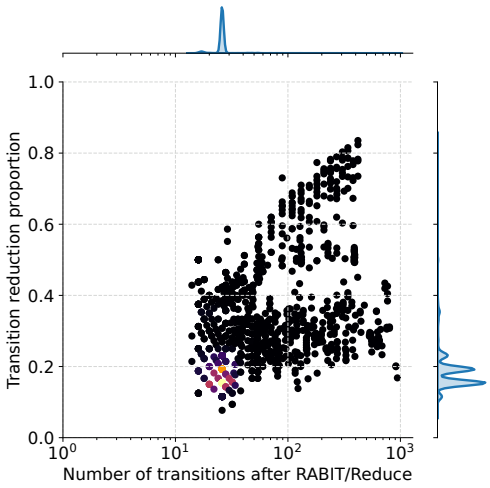
1. formulae derived from the WS1S-based shape analysis of [30] that contain various invariants over structures like single-linked lists or the bubble sort algorithm,
2. formulae from [43], reasoning about programs with unbounded arrays,
3. parametric formulae that come from the work of D'Antoni et al. [15], artificially enhanced by added quantifier alternations,
4. parametric formulae used to evaluate the Toss tool as presented in [22], and
5. parametric formulae from the evaluation of the dWiNA tool [19].

×	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Map}+\Gamma_{Map}$		δ_{Map}		Γ'_{Map}
Average	14	28	14	28	8+2	(71.4%)	22	(78.6%)	3
Best Q and δ	399	419	399	419	27+30	(14.3%)	69	(16.5%)	20

Table 6.9: Average and the best reduction results in the number of states (Best Q) and transitions (Best δ) after applying RABIT/Reduce (Q_{RAB}, δ_{RAB}) and the Procedure mapping ($Q_{Map}+\Gamma_{Map}, \delta_{Map}$). Γ_{Map} is the stack alphabet and Γ'_{Map} is the minimal stack alphabet calculated by *reduceStack* algorithm. The percentages refer to RABIT/Reduce results.



The reductions in the number of states ($Q_{Map}+\Gamma_{Map}$) after applying Procedure mapping to the results of RABIT/Reduce.



The reductions in the number of transitions after applying Procedure mapping to the results of RABIT/Reduce.

Figure 6.8: The reductions, for automata created from benchmarking Pavel Bednář's tool, after applying the Procedure mapping algorithm to the results from the RABIT/Reduce tool. On average, we achieved a reduction of 28.6% in states and 21.4% in transitions.

The evaluation was performed on 40,535 automata with an average of 14 states and 28 transitions, and with a maximum of 521 states and 800 transitions. The average and

best reduction results are summarized in Table 6.9. The reduction tool RABIT/Reduce was not able to perform any visible reduction on these automata at all. However, the application of the Procedure mapping algorithm resulted, on average, in 28.6% reduction in states and 21.4% reduction in transitions. The results of all automata reductions are illustrated in Figure 6.8.

The best reduction achieved by our Procedure mapping algorithm was achieved on the automaton from the third family of formulae. Initially, the automaton contained 399 states and 419 transitions. Similarly to previous cases, RABIT/Reduce did not perform any reduction. However, the Procedure mapping algorithm reduced the automaton to 27 states with 30 stack symbols, resulting in 85.7% reduction in the number of states and 69 transitions, representing 83.5% reduction in the number of transitions.

It is worth mentioning that due to the presence of many small automata in the benchmarking set, the average reductions achieved by the Procedure mapping algorithm tends to be small. However, as illustrated in Figure 6.8, the reduction of states and transitions increases with the size of the input automata. This indicates that the Procedure mapping algorithm is highly suitable even for these kinds of automata.

Chapter 7

Conclusion

As the size of nondeterministic finite automata plays an important role in the complexity of algorithms that operate on them, it is crucial to reduce their size. Although there are many methods for minimizing NFAs, such as state merging, transition pruning, or saturation, they often leave redundant transition sequences in the automaton. There are even types of automata, often representing regular expressions, that cannot be reduced at all.

In this work, we presented a novel approach for NFAs that can reduce the size of the automaton by transforming it into an equivalent NPDA_1. By doing so, we can efficiently eliminate repeating substructures in the automata. More specifically, we replace the redundant transition sequences in the NFA with a single procedure and store the information about the state from which it was entered and the state to which it should return in the form of one stack symbol. This transformation can be understood as a conversion from a purely sequential program into a program with functions that use a call stack to store information about the point in the program where it should return.

We evaluated our approach, implemented in Python, as an additional step after the standard minimization tool RABIT/Reduce, implemented in Java, which utilized state merging and transition pruning. We tested our approach on a diverse set of 52,968 automata, including automata representing various regular expressions or string constraints, automata generated during Abstract Regular Model Checking, by the Z3-noodler tool, or during the decision of formulae in WS1S. On average, we achieved a 29.5% reduction in the number of states and a 30.9% reduction in the number of transitions. Additionally, we tested the Procedure mapping algorithm on a set of automata created from regular expressions used within the rules of the network intrusion detection system Snort. The reduction of these automata is crucial for the performance of the NIDS, as it can significantly speed up the process of scanning network traffic. On these automata, we achieved a reduction of up to 44.6% in the number of states and 60.3% in the number of transitions compared to the results of standalone usage of the RABIT/Reduce tool. The most significant reduction was achieved on automata generated during the computation of the Z3-noodler tool, where we reached a reduction of 64.9% in states and 67.3% in transitions.

The experiments demonstrated that our minimization method is suitable not only for automata representing regular expressions but also for various other types of automata, as significant reductions in the number of states and transitions were consistently achieved. This suggests that our approach could be utilized in the future as a general reduction method, complementing state merging, transition pruning, and saturation, for more efficient minimization of NFAs.

In future work, our aim is to enhance our approach by reducing its time complexity and making it more suitable for larger automata with thousands of states. Furthermore, we plan to implement NPDA_1 operations such as intersection, union, minimization, determinization, and complementation in C++ to further improve the performance of our approach. Additionally, we plan to apply our approach to the minimization of NFAs for high-speed network filtering and integrate these automata into hardware for real-time processing of network traffic. This will present challenges in the field of on-the-fly determinization of automata in hardware, where determinization must consider not only the state of the automaton, as usual, but also the symbol that can appear on the stack.

Bibliography

[1] Abdulla, P. A.; Atig, M. F.; Holík, L.; Chen, Y.-F.; Rezine, A. et al. *Norn: An SMT Solver for String Constraints*. 2015. Available at: https://user.it.uu.se/~jarst116/norn/.

[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] Abdulla, P. A.; Chen, Y.-F.; Holík, L.; Mayr, R. and Vojnar, T. When Simulation Meets Antichains. In: Esparza, J. and Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 158–174. ISBN 978-3-642-12002-2.

[4] 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.

[5] 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.

[6] Bednář, P. *Deciding WS1S with Automata*. 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Holík, L. Available at: https://www.vut.cz/studenti/zav-prace/detail/144803.

[7] 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. ISSN 1433-2787.

[8] 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.

[9] Ceška, M.; Havlena, V.; Holík, L.; Korenek, J.; Lengál, O. et al. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, p. 109–117.

[10] Chomsky, N. *Context-Free Grammars and Pushdown Storage*. Quarterly Progress Report 65. MIT Research Laboratory of Electronics, 1962.

[11] Clemente, L. and Mayr, R. Advanced Automata Minimization. *CoRR*, 2012, abs/1210.6624.

[12] Clemente, L. and Mayr, R. Efficient Reduction of Nondeterministic Automata with Application to Language Inclusion Testing. *CoRR*, 2017, abs/1711.09946.

[13] Clemente, L.; Mayr, R. et al. *RABIT/Reduce*. https://languageinclusion.org.

[14] D'Antoni, L.; Kincaid, Z. and Wang, F. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *CoRR*, 2016, abs/1610.01722.

[15] D'Antoni, L. and Veanes, M. Minimization of Symbolic Automata. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2014, p. 541–553. POPL '14. ISBN 9781450325448.

[16] 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.

[17] Evey, R. J. Application of Pushdown-Store Machines. In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. New York, NY, USA: Association for Computing Machinery, 1963, p. 215–227. AFIPS '63 (Fall). ISBN 9781450378833.

[18] Fiedor, T.; Holik, L.; Janku, P.; Lengal, O. and Vojnar, T. *Gaston - Decision Procedure for WS1S Logic*. Available at: https://www.fit.vutbr.cz/research/groups/verifit/tools/gaston/.cs.

[19] Fiedor, T.; Holík, L.; Lengál, O. and Vojnar, T. Nested Antichains for WS1S. In: Baier, C. and Tinelli, C., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, p. 658–674. ISBN 978-3-662-46681-0.

[20] Fiedor, T.; Holík, L.; Lengál, O. and Vojnar, T. Nested Antichains for WS1S. *Acta Informatica*, 2019, vol. 56, no. 3, p. 205–228.

[21] Gange, G.; Navas, J. A.; Stuckey, P. J.; Søndergaard, H. and Schachte, P. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In: Piterman, N. and Smolka, S. A., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 277–291. ISBN 978-3-642-36742-7.

[22] Ganzow, T. and Kaiser, Ł. New Algorithm for Weak Monadic Second-Order Logic on Inductive Structures. In: Dawar, A. and Veith, H., ed. *Computer Science Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 366–380. ISBN 978-3-642-15205-4.

[23] Ginsburg, S. The Mathematical Theory of Context Free Languages. *Journal of Symbolic Logic*. Association for Symbolic Logic, 1968, vol. 33, no. 2, p. 300–301.

[24] Havlena, V.; Síč, J.; Chen, Y.-F.; Chocholatý, D.; Holík, L. et al. *Z3 Noodler*. Available at: https://github.com/VeriFIT/z3-noodler.

[25] Henzinger, M.; Kopke, P. and Henzinger, T. Computing Simulations on Finite and Infinite Graphs. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct 1995, p. 453. ISSN 0272-5428.

[26] Hopcroft, J. E.; Motwani, R. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

[27] 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. ISBN 978-3-540-27812-2.

[28] Korenek, J. and Kobiersky, P. Intrusion Detection System Intended for Multigigabit Networks. In: *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems*. 2007, p. 1–4.

[29] Kreowski, H.-J. A Pmping Lemma for Context-Free Graph Languages. In: Claus, V.; Ehrig, H. and Rozenberg, G., ed. *Graph-Grammars and Their Application to Computer Science and Biology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, p. 270–283. ISBN 978-3-540-35091-0.

[30] Madhusudan, P.; Parlato, G. and Qiu, X. Decidable Logics Combining Heap Structures and Data. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2011, p. 611–622. POPL '11. ISBN 9781450304900.

[31] Pus, V.; Tobola, J.; Kosar, V.; Kastil, J. and Korenek, J. Netbench: Framework for Evaluation of Packet Processing Algorithms. In:. October 2011, p. 95–96.

[32] Rabin, M. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, april 1959, vol. 3, p. 114–125.

[33] RegexLib. *Regular Expression Library*. Available at: https://regexlib.com/.

[34] SMT-LIB Team. *Benchmarks*. http://smtlib.cs.uiowa.edu/benchmarks.shtml.

[35] SMT-LIB Team. *SMT-LIB: QF_S*. Available at: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S.

[36] SMT-LIB Team. *SMT-LIB: QF_S/20230329-automatark-lu*. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S/-/tree/master/20230329-automatark-lu.

[37] Snort Team. *Snort: Network Intrusion Detection & Prevention System*. Available at: https://www.snort.org/.

[38] 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.

[39] Stanford, C.; Veanes, M. and Bjørner, N. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 620–635. PLDI 2021. ISBN 9781450383912.

[40] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972, vol. 1, no. 2, p. 146–160.

[41] Vargovčík, P. and Holík, L. Simplifying Alternating Automata for Emptiness Testing. In: Oh, H., ed. *Programming Languages and Systems*. Cham: Springer International Publishing, 2021, p. 243–264. ISBN 978-3-030-89051-3.

[42] Šedý, M. *Reducing Size of Nondeterministic Automata with SAT Solvers*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Supervisor Holík, L. Available at: https://www.fit.vut.cz/study/thesis/23436/.

[43] Zhou, M.; He, F.; Wang, B.-Y.; Gu, M. and Sun, J. Array Theory of Bounded Elements and its Applications. *Journal of Automated Reasoning*, Apr 2014, vol. 52, no. 4, p. 379–405. ISSN 1573-0670.

[44] Češka, M.; Havlena, V.; Holík, L.; Lengál, O. and Vojnar, T. *AppReAL - Approximate Reduction of Automata and Languages*. Available at: https://github.com/ondrik-network-hw/appreal.

Appendix A

Excel@FIT 2024

In this apendix, we will present the poster and its commentary that were successfully accepted at the Excel@FIT 2024 conference. The poster was presented at the conference, and the commentary was published together with the poster in the conference proceedings available at https://excel.fit.vutbr.cz/2024/sbornik. The poster was awarded by the conference committee for its innovative approach to reducing the memory complexity of automata representation.

Repetitive Substructures for Efficient Representation of Automata

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Michal Šedý

supervised by doc. Mgr. Lukáš Holík, Ph.D.

Excel@FIT 2024

Motivation

In many automata, especially those representing regular expressions, there exist repetitive substructures that cannot be eliminated using the state-of-the-art tool RABIT/Reduce [2]. This automaton is depicted in Figure 1 below.

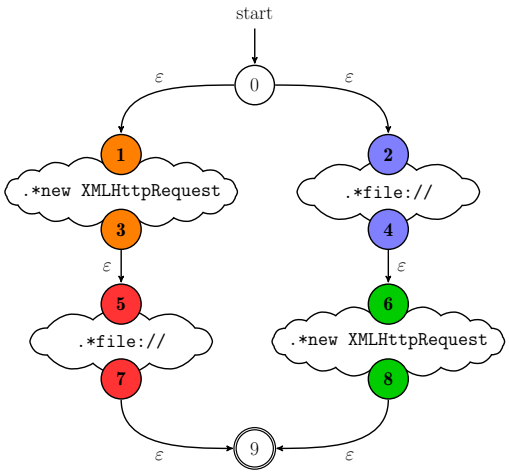


Fig. 1: Automaton with duplicate substructures.

We propose a novel approach based on pushdown automata and so-called procedures, which represent repetitive substructures only once.

Usage of a pushdown automaton and procedures is analogous to the call stack and functions from programming languages.

One Procedure No Duplicates

To represent automata efficiently, without duplicate substructures, we introduce a new concept called procedures. Each set of similar substructures is represented by one procedure. The automaton uses a stack to determine the state from which the procedure is entered and the state to which it should return. The symbol on the stack can also serve to guard transitions that are specific to certain substructures represented by the procedure.

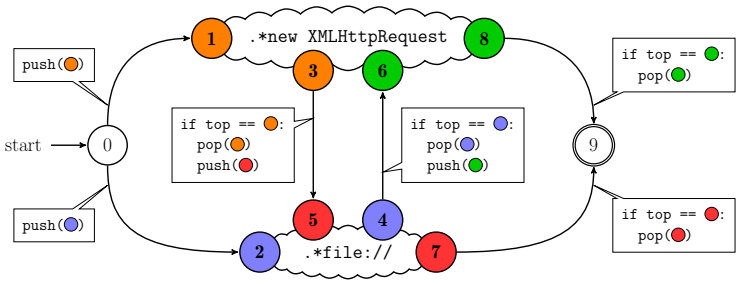


Fig. 2: Automaton with two procedures and no duplicate substructures.

Parametric Regular Expressions

We evaluated the reduction potential of procedures on 3'656 automata, with an average of 207 states and 2'584 transitions, generated from parametric regular expressions [1].

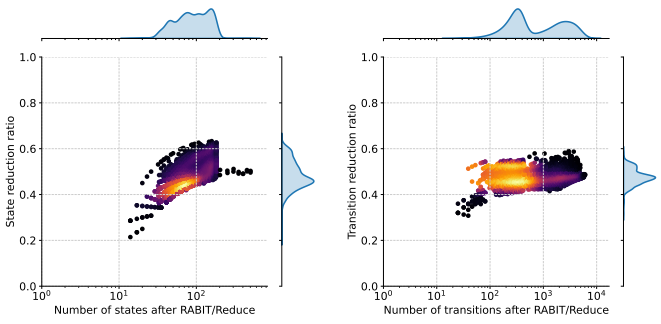


Fig. 3: The reduction ratios achieved by applying procedures to RABIT/Reduce results. On average, procedures improved reductions by 50.3% in states and 47.9% in transitions.

The standalone usage of RABIT/Reduce resulted on average in a reduction of 52.5% in states and 48.4% in transitions. The further reduction performed by our algorithm can be seen in Figure 3. The application of procedures reduced the automata to half the size given by RABIT/Reduce.

Network Intrusion Detection System

To test the reduction capability of procedures in a real-world scenario, we used rules from Snort (a well-known NIDS). We generated seven automata, each representing a union of regular expressions, from seven different categories of Snort rules.

Snort rules	Q_{in}	δ_{in}	Q_{RAB}	δ_{RAB}	$Q_{Proc}+\Gamma_{Proc}$	δ_{Proc}
p2p	33	1'090	32	1'084	25+6 (96.9%)	570 (52.6%)
worm	50	3'880	34	290	24+8 (94.1%)	284 (97.9%)
shellcode	162	3'328	56	579	48+2 (89.3%)	486 (83.9%)
mysql	235	30'052	91	14'430	45+18 (69.2%)	7'142 (49.5%)
chat	408	23'937	113	1'367	71+25 (85.0%)	1'058 (77.4%)
specific-threats	459	57'292	236	31'935	99+32 (55.5%)	12'680 (39.7%)
telnet	829	7'070	309	2'898	155+82 (76.7%)	2'164 (74.7%)

Tab. 1: Reduction results of RABIT/Reduce (RAB) and procedures (Proc) on seven sets of Snort rules. Q denotes the number of states δ the number of transitions, and Γ the number of stack symbols. The percentages refer to the results of RABIT/Reduce.

Among the reduction results in Table 1, we highlighted the two most significant reductions. The best size reduction was achieved on the `specific-threats` rule. RABIT/Reduce tool reduced the automaton by 48.6% of states and by 44.3% of transitions. Further application of procedures resulted in an additional reduction of 44.5% in states and 60.3% in transitions. This experiment showed that procedures can achieve significant reductions even in real-world examples.

References

[1] Gange, G. et al. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In: *TACAS'13*. Springer, 2013. ISBN 978-3-642-36742-7.

[2] Mayr, R. et al. *RABIT and Reduce*. `https://languageinclusion.org`.

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2024
http://excel.fit.vutbr.cz

Repetitive Substructures for Efficient Representation of Automata

Michal Šedý*
supervised by doc. Mgr. Lukáš Holík, Ph.D.

Abstract
Nondeterministic finite automata are widely used across almost every field of computer science, such as for the representation of regular expressions, in network intrusion detection systems for monitoring high-speed networks, in abstract regular model checking, program verification, or even in bioinformatics for searching sequences of nucleotides in DNA. To obtain smaller automata, and thus reduce computational resources, state-of-the-art minimization techniques, such as state merging and transition pruning, are used. However, these methods can still leave duplicate substructures in the resulting automata. This work presents a novel approach to automata minimization based on the transformation of an NFA into a nondeterministic pushdown automaton. The transformation identifies and represents similar substructures only once by so-called procedures. By doing so, we can further reduce automata by up to 60%.

*xsedym02@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Nondeterministic finite automata (NFAs), as their name suggests, can nondeterministically transition from one state to multiple states based on the same input. This property allows NFAs to represent languages more compactly than their deterministic counterparts, which can only be in one state at a time. Despite their hard minimization, NFAs find applications in many fields of computer science, such as representing regular expressions, network intrusion detection systems for monitoring high-speed networks [1, 2], abstract regular model checking [3], verifying programs that manipulate strings [4], or decision procedures in the WS1S and WS2S logic [5, 6].

Minimization Techniques

To reduce computational resources when working with NFAs, it is crucial to reduce their size. For this purpose, the state merging [7, 8, 9] and transition pruning [8, 10] techniques are being used. The state merging technique can merge two states if one of them fully covers the logic of the other. On the other hand, transition pruning removes a transition if there already exists a duplicate transition with the same logic. These methods are implemented in the state-of-the-art tool RABIT/Reduce [11].

Repetitive Substructures

Despite the good reduction potential that standard minimization techniques offer, the resulting automata can still contain redundant substructures. These automata often represent regular expressions, such as those used in network intrusion detection systems (NIDSs) for network traffic scanning. They are constructed as the union of regular expressions. Additionally, there are types of automata that cannot be minimized by these standard methods at all.

Our Novel Approach

In our work, we present a novel reduction approach that involves transforming a NFA into a nondeterministic pushdown automaton (NPDA) that utilizes a stack. This approach identifies similar substructures within the automaton and represents them only once using so-called procedures. The stack is then utilized to track the states from which the procedure has been entered and where to return. This transformation can be likened to converting a purely sequential program into one that uses functions and a call stack. By applying our approach to the results of standard minimization techniques, we were able to achieve an additional reduction of up to 60% in both the states and the transitions of the automaton.

2. Motivation

The automaton representing a regular expression `(.*new XMLHttpRequest.*file://)|(.*file://.*new XMLHttpRequest)` from network intrusion detection system Snort [12] is shown in Figure 1. Besides the epsilon transitions and `.*`, this is the most minimal form that can be achieved by standard minimization techniques. This is caused by the lack of language inclusions as *Request* and *File* substructures are completely different. As a result, the automaton contains two substructures, each of which has redundant copy, making the NFA representation unefficient.

3. One Procedure No Duplicate

At this point, we identified repetitive substructures for *Request* and *File*, which represent duplicate information. Each of these substructures will be represented by a corresponding procedure. This transformation into procedures can be seen in Figure 2.

Entering the Procedure

To recognize if the procedure for *Request* has been entered from state 1 as the first part of the regular expression or from state 6 as the second part, the symbol 🟠 or 🟢 is pushed onto the stack, respectively.

Changing Procedures

When directly transferring from the *Request* procedure into the *File* procedure, it is necessary to ensure that this transition will be used only once and only after first entering the *Request* from state 1. This is done by testing for the symbol 🟠 at the top of the stack. If the top matches the required symbol, it is replaced with the symbol 🔴. The same approach applies for the transition from the *File* procedure into the *Request* procedure.

Returning From the Procedure

Transitions exiting the *Request* procedure can only be used when the stack contains the corresponding symbol that is popped afterward. For the transition between states 8 and 9, it is the symbol 🟢, indicating that the *Request* procedure represents the last part of the regular expression that started with the *File*.

4. Experiments

We tested our reduction method on parametric and real-world regular expressions used in network filtering. The highest reductions were obviously achieved on larger automata, as there is a greater likelihood of similar substructures existing. Since our tool is designed to follow after standard reduction methods, the percentage reduction is calculated relative to the resulting automata of RABIT/Reduce.

4.1 Parametric Regular Expressions

The set of 3'656 automata, with an average of 207 states and 2'584 transitions, was obtained from four families of parametric regular expressions [13]. The reduction ratio of states and transitions can be seen in Figure 3. It can be observed that, on average, our tool achieved a reduction of 52.5% in the number of states. The x-axis of the graph represents the size of RABIT/Reduce results (the input of our tool), while the y-axis represents achieved reduction ratio. The graph is enhanced with temperature coloring and a distribution function for each axis.

4.2 Network Intrusion Detection System

To test our reduction algorithm on real-world examples, seven automata were created as unions of sets of regular expressions from seven different families of Snort rules. The results of reduction performed by standalone usage of the RABIT/Reduce tool (*RAB*) and with the additional application of our method based on procedures (*Proc*) are shown in Table 1. The two most significant results are highlighted. The best obtained result achieved a reduction of 44.5% in states and 60.3% in transitions.

5. Conclusion

In this work, we introduced a novel approach to automata reduction. This reduction transforms NFAs into NPDAs, noting the similarity with transforming a pure sequential program into a program with functions and call stack. Applying our reduction approach to the results of the state-of-the-art reduction tool RABIT/Reduce resulted in reductions of up to 52.5% in states and up to 60.3% in transitions. These results suggest that our approach could significantly contribute to the reduction of automata in the future.

References

[1] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system. In *FPL'03*. Springer, 2003.

[2] M. Ceška, V. Havlena, L. Holík, et al. Deep packet inspection in FPGAs via approximate nondeterministic automata. In *FCCM'19*, 2019.

[3] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *STTT'12*, Apr 2012.

[4] P. A. Abdulla, M. F. Atig, Y.F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV'14*. Springer, 2014.

[5] C. Fu, Y. Deng, D. Jansen, and L. Zhang. On equivalence checking of nondeterministic finite automata. In *SETTA'17*. Springer, 2017.

[6] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. Nested antichains for WS1S. *Acta Informatica*, 2019.

[7] A. Aziz, V. Singhal, R. Brayton, and G.M. Swamy. Minimizing interacting finite state machines: a compositional approach to language containment. In *ICCD'94*, 1994.

[8] D. Bustan and O. Grumberg. Simulation based minimization. In D. McAllester, editor, *CADE'20*. Springer.

[9] L. Ilie, G. Navarro, and S. Yu. *On NFA Reductions*. Springer, 2004.

[10] L. Clemente and R. Mayr. Efficient reduction of nondeterministic automata with application to language inclusion testing. *CoRR*, 2017.

[11] R. Mayr, L. Clemente, et al. RABIT and Reduce. https://languageinclusion.org.

[12] Snort Team. Snort - Network Intrusion Detection & Prevention System. https://snort.org.

[13] G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS'13*. Springer, 2013.