BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ABSTRACTION OF STATE LANGUAGES IN AUTOMATA ALGORITHMS

ABSTRAKCE JAZYKŮ STAVŮ V AUTOMATOVÝCH ALGORITMECH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DAVID CHOCHOLATÝ**
AUTOR PRÁCE

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

BRNO 2022

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

Bachelor's Thesis Specification

25013

Student: **Chocholatý David**
Programme: Information Technology
Title: **Abstraction of State Languages in Automata Algorithms**
Category: Algorithms and Data Structures

Assignment:

The goal is to explore possibilities of using various abstractions of automata languages in optimisation of automata algorithms.
We will start with abstracting languages of states to sets of possible word lengths and to Parikh images, represented as semi-linear sets, and exploring options of using them to optimize the construction of synchronous product of automata by pruning pairs of states with incompatible abstractions. We will then continue either towards optimisation of these techniques or towards searching for alternatives or more advanced versions.

1. Study the literature on automata and Parikh images, familiarise yourself with technology of SMT solvers.
2. Implement automata product construction optimised with length abstraction and Parikh image abstraction of state languages.
3. Study the state pruning capabilities of these abstractions.
4. If the pruning capabilities prove promising, study possibilities of their efficient implementation and compare it with standard synchronous product construction on provided data.
5. Otherwise continue the research by elaborating on the principle of using state language abstractions to optimize automata constructions.

Recommended literature:

- Javier Esparza, Pierre Ganty, Stefan Kiefer, Michael Luttenberger: Parikh's Theorem: A simple and direct construction. CoRR abs/1006.3825, 2010.

Requirements for the first semester:

- Items 1 and 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: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

We explore possibilities of using various abstractions of finite automata languages in optimization of automata algorithms used in automata reasoning. We focus on abstracting languages of states to sets of accepted lengths of word or Parikh images, represented as semi-linear sets, and exploring options of using them to optimize automata constructions by pruning states based on abstractions of their languages. We propose several abstractions and work on optimizing their performance. We use two common finite automata problems, synchronous product construction and deciding the emptiness of finite automata intersection, as benchmark problems on which we test our optimizations. Nevertheless, our abstractions are applicable on many other typical automata operations, e.g., complement generation etc. Our experiments show that the proposed optimizations reduce generated state space for both benchmark problems substantially.

Abstrakt

Prověřujeme možnosti použití různých abstrakcí jazyků konečných automatů pro optimalizaci automatových algoritmů používaných pro rozhodování založeném na automatech. Zajímáme se o abstrakci jazyků stavů na množiny přijímaných délek slov nebo Parikovy obrazy, reprezentované jako semi-lineární množiny, a zkoumáme možnosti jejich využití k optimalizaci automatových konstrukcí odstraňováním stavů založené na abstrakcích jejich jazyků. Předvádíme několik abstrakcí a pracujeme na optimalizaci jejich chování. Používáme dva běžné automatové problémy, synchrnonní produkt construkci a rozhodování prázdnosti průniku konečných automatů, jako operace pro experimentální vyhodnocení, na kterých testujeme naše optimalizace. Naše abstrakce jsou nicméně aplikovatelné na mnohé další typické automatové operace, například generaci doplňku aj. Provedené experimenty ukazují, že navrhované optimalizace podstatně zmenšují generovaný stavový prostor pro oba testované problémy.

Keywords

finite automata, state language abstractions, SMT solving, product construction, emptiness test, intersection computation optimization, state space reduction, length abstraction, Parikh images, mintermization

Klíčová slova

konečné automaty, abstrakce jazyků stavů, SMT výpočty, konstrukce produktu, test prázdnosti, optimalizace výpočtu průniku, redukce stavového prostoru, délková abstrakce, Parikovy obrazy, mintermizace

Reference

CHOCHOLATÝ, David. *Abstraction of State Languages in Automata Algorithms.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

Konečné automaty nachází mnohá využití v oblastech, jako jsou logika (WS1S), matematika nebo výpočetní teorie (model checking nebo string solving a analýza). Přestože jsou konečné automaty konceptuálně jednoduché, často s nimi potřebujeme provádět operace, které jsou výpočetně drahé a generují rozsáhlý stavový prostor, jehož mnohé části jsou nadbytečné.

V této práci zkoumáme možnosti použití různých abstrakcí jazyků stavů automatů pro optimalizaci takových automatových algoritmů. Pomocí vhodných abstrakcí se snažíme předpovědět, které stavy výsledného automatu jsou nepotřebné, a mohou proto být odstraněny z generovaného stavového prostoru bez narušení jazyku výsledného automatu, pokud jsou získané abstrakce navzájem nekompatibilní.

Pro demonstraci našich abstrakcí jsme se rozhodli použít operaci průniku konečných automatů prováděnou synchronní produkt konstrukcí a test prázdnosti průniku automatů. Naše předvedené abstrakce jsou však navrženy tak, aby byly aplikovatelné na širokou škálu automatových operací (například konstrukci doplňku aj.). Význam naší práce proto přesahuje samotnou optimalizaci produkt konstrukce automatů. Všechny navrhované abstrakce navíc tvoří *Galois connection*, tedy popisují nad-abstrakci jazyků stavů. Díky tomu není nebezpečí, že bychom při odstraňování stavů s nekompatibilními abstrakcemi nechtěně odstranili i stavy důležité pro popis jazyka přijímaného generovaným automatem.

Při konstrukci průniku automatů dochází k tzv. stavové explozi, kdy jsou generovány rozsáhlé části stavového protoru, které tvoří neukončující stavy, ze kterých nebude dosažitelný žádný koncový stav ve výsledném produktu. Naše optimalizace sestává z kontroly kompatibility abstrakcí jazyků stavů pro stavy, ze kterých se skládá daný produkto-stav, za běhu produkt konstrukce. Pokud určíme abstrakce jako nekompatibilní, můžeme bezpečně takový produkto-stav odstranit. Výhodou našich abstrakcí je, že stavový prostor zmenšují již při generaci výsledného automatu. Některé stavy tak nebude třeba vůbec ani generovat, pokud všechny jejich předchůdci budou odstraněni. Naproti tomu u naivního algoritmu produkt konstrukce musíme nejdříve vygenerovat celý automat, než můžeme rozhodovat o kompatibilitě jazyků vstupních automatů.

Mezi zkoumané abstrakce jazyků stavů patří délková abstrakce a abstrakce Parikovými obrazy. Dále zkoumáme možnosti optimalizace těchto abstrakcí či předzpracování vstupních automatů, například pomocí mintermizace automatů.

Délková abstrakce abstrahuje jazyk stavů na lineární množiny možných délek slov přijímaných jazykem pomocí lineárních délkových formulí. Aby daný produkto-stav patřil do průniku, přijímané délky slov stavů ve vstupních automatech si musí odpovídat, tedy formule popisující délkovou abstrakci musí být splnitelné zároveň. V opačném případě jazyky stavů nepřijímají stejný jazyk (délky přijímaných slov se liší) a jejich průnik je prázdný. Takové produkto-stavy mohou být odstraněny z generovaného stavového prostoru a jejich následníci nemusí být generováni.

Délky slov modelujeme pomocí tzv. laso automatů přijímajících nadmnožinu jazyka vstupních automatů přijímající slova o všech délkách slov přijímaných vstupními automaty. Vzájemnou splnitelnost délkových abstrakcí v podobě délkových formulí sestavených z laso automatů ověřujeme zadáním příkazu pro SMT solver, nicméně můžeme optimalizovat otázku splnitelnosti délkových formulí nahrazením SMT solveru za matematický výpočet založený na vlastnostech lineární kongruence, který je schopný rychle a efektivně rozhodnout o splnitelnosti délkových formulí.

Abstrakce Parikovými obrazy definuje semi-lineární množiny založené na Parikově teorému abstrahující jazyky stavů na počty výskytů symbolů na přechodech bez závislosti na jejich umístění v přijímaném slově pomocí semi-lineárních formulí Parikových obrazů. Za nekompatibilní abstrakce považujeme takové, kde si neodpovídají počty použitých symbolů jazyků stavů pro daný produkto-stav. Tedy, pokud jsou formule Parikových obrazů navzájem nesplnitelné, můžeme opět odstranit daný produkto-stav z generovaného stavového prostoru.

Abstrakci Parikovými obrazy je možné nadále optimalizovat další redukcí Parikových obrazů či inkrementálním SMT výpočtem, který umožňuje předpočítat společné části formulí jednou a využívat po celou generaci produktu. Nadále můžeme zavést *timeout* pro předčasné ukončení rozhodování splnitelnosti formulí Parikových obrazů.

Obě abstrakce mohou využít optimalizace přeskočitelných produkto-stavů, kdy není třeba vyhodnocovat splnitelnost formulí abstrakcí, pokud daný produkto-stav byl vytvořen z produkto-stavu generujícího pouze tento jediný následující produkto-stav. Tedy, aby měl předcházející produkto-stav kompatibilní abstrakce jazyků stavů vstupních automatů, musí využívat aktuálního produkto-stavu pro dosažení koncového stavu, a proto musí nutně i abstrakce jazyků stavů pro tento následující produkto-stav být navzájem kompatibilní.

Naše abstrakce jsme navrhli tak, aby tvořily obecný a samostatný popis jazyků stavů, což umožňuje abstrakce volitelně kombinovat, rozšiřovat o další abstrakce či optimalizační techniky, a tím využít výhod každé abstrakce, zatímco minimalizujeme dopad nevýhod daných abstrakcí. Tím umožňujeme využívat naše optimalizace pro širokou oblast problémů řešených konečnými automaty. Přístup za běhu řešených abstrakcí jazyků stavů taktéž umožňuje operace paralelizovat nebo vhodně rozdělit na podproblémy.

Provedli jsme experimentální vyhodnocení navrhovaných abstrakcí optimalizujících konstrukci průniku. Podle provedených experimentů můžeme soudit, že navrhované abstrakce mají předpokládané optimalizační schopnosti a zmenšují generovaný stavový prostor i lépe rozhodují test prázdnosti průniku automatů než naivní přístupy konstrukce produktu.

Délková abstrakce je rychlá a efektivní, její optimalizační síla je však nižší než u abstrakce Parikovými obrazy. Délková abstrakce výborně optimalizuje produkty s dlouhými linkami stavů, může mít však potíže s odstraňováním stavů v hustě propletené síti přechodů. Abstrakce Parikovými obrazy naopak skvěle optimalizuje generovaný produkt, ovšem výpočet vzájemné splnitelnosti formulí Parikových obrazů je pro SMT solver náročný a časově drahý. Můžeme si tedy zvolit, jestli chceme dosáhnout rychlého, i když možná méně důkladného zmenšení stavového prostoru; přesné, ale výpočetně náročnější minimalizace průniku; případně vhodné kombinace těchto vlastností.

Abstraction of State Languages in Automata Algorithms

Declaration

I hereby declare that this Bachelor'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.

....
David Chocholatý
May 10, 2022

Acknowledgements

I would like to thank my supervisor, doc. Mgr. Lukáš Holík, Ph.D., who has provided essential and necessary information about the topic, outlined possible solutions and answered every question I have had throughout the whole time.

Contents

Chapter 1

Introduction

Finite automata are a well-known model of computational theory used in many areas. Finite automata are commonly used in automata reasoning (e.g., in model checking [26] or string solving and analysis [24]). Their usage in the field of logic is just as important, too (e.g., WS1S [13, 14]). Finite automata are conceptually straightforward. However, such operations on finite automata are often expensive: have high complexity, require extensive computational time and generate vast state space.

Our goal is to find different heuristics for optimizing several typical problems connected to finite automata. We study possibilities of using various abstractions of languages of finite automata states in optimization of automata algorithms. We abstract languages of specific states to sets of lengths of word in finite automata language and to Parikh images, represented as semi-linear sets, and explore options of using them to optimize the automata constructions by pruning states whose language abstractions represent an empty language. We work on optimizing performance of these abstractions. Moreover, besides optimization techniques specific for concrete state language abstractions, we also consider a general technique of mintermization and other general approaches to optimize our abstractions further.

We consider several usual finite automata operations which take lots of computational time and generate vast state space. One such operation is the construction of finite automata intersection generated by the synchronous product construction. We consider two common forms of this finite automata operation as our benchmark problems on which we test our optimizations:

- first, construction of the intersection automaton by the synchronous product construction, which means completion of the entire product construction, and
- testing the emptiness of finite automata intersection (emptiness problem). Here, it is not always necessary to construct the entire product (or parts of the product) to resolve the emptiness of the intersection.

Nevertheless, even if our optimizations are introduced on product construction and emptiness problem, our discoveries have wider impact and are in some form applicable on many typical automata operations, e.g., determinization of complement construction.

The intersection of finite automata, an extensively used automata operation, combines the original states from the individual automata to tuples called product states in the generated state space by finding corresponding transitions with the same symbols. Every product state represents an intersection of languages of two corresponding states in the original automata. The synchronous product construction is computationally costly: for two finite automata, the generated product state space can increase quadratically to the number of input finite automata states (number of states in one finite automaton times number of states in the second finite automaton) and transitions. And, for multiple finite automata, exponentially to the number of used finite automata. However, there are often large parts of the generated state space which cannot accept any words (no final states can be reached from these states), yet are still generated[1]. Therefore, it is important to have a decent algorithm to minimize the generated product state space as much as possible.

In our optimizations, we try to identify which generated product states cannot lead to any accepting state or are successive only to such states. When state language abstractions of states in product state are not compatible—the original languages of the corresponding states cannot accept the same words—we can omit such product state and all their potential successive states, pruning the generated state space.

We start with an optimization using length abstraction of state languages. For each state, we construct a so-called lasso automata. It is used to compute a semi-linear formula which codes the lengths of the words in the languages of current states (we call them accepted lengths). We use SMT solver to resolve satisfiability of these formulae. When accepted lengths of states in the product states are not compatible (their formulae are not satisfiable), their languages have no common words. There is no path from the product state leading to the accepting product state. We can prune such product states. Consequently, this removes the need to even consider their potential successive states.

Even though there still might be states which do not lead to any final state in the final product, this simple optimization often trims substantial parts of the state space. Length abstraction can also be implemented simply and efficiently. However, sometimes the abstraction is too coarse. For instance, it cannot detect unnecessary product states for finite automata with rich alphabets, since their states accept a multitude of word lengths.

For that reason, we investigate a finer state language abstraction which uses Parikh images of state languages. Parikh image of a word tells us how many times each symbol occurs in the word[2]. Parikh image of a language is a semi-linear formula describing the relation between the number of symbol occurrences in words in a language. In contrast to the length abstraction, it contains additional information about the numbers of symbols in words. We can more precisely identify unnecessary state space by determining compatibility of Parikh image abstractions. To solve satisfiability of their formulae, we use SMT solver. However, the Parikh image computation is expensive. There is a trade-off between the precision of the Parikh image abstraction and its cost.

Generating smaller state space using our Parikh image optimization can improve computation time for the product generation in case substantial parts of the state space are pruned. Moreover, it is even enough to decide the intersection emptiness on its own.

An important part of this work is researching optimizations of the specific state language abstractions to make their usage efficient. For both abstractions, we find a way to skip evaluation of state language abstractions for product states in long *lines* (linear non-branching sequences of states). If a product state with compatible accepted lengths generates a single sequence of states, all states in the line have compatible length abstractions.

For length abstraction, we reduce lasso automata generation for each state to a single expanding lasso automaton for the whole finite automaton. We also efficiently evaluate

[1] The generated product state space sometimes *explodes*.

[2] A function which assigns each transition symbol a number of occurrences in a word.

length abstractions without SMT solver by resolving satisfiability of their formulae with a special construction using linear congruences.

For Parikh image computation, we remove parts of Parikh image formula which reduce its precision, but it occurs that it has no impact on pruning capabilities. The formulae are large. However, extensive parts of them remain unchanged for different product states. We utilize incremental SMT solving to precompute the common parts and for each state recompute only the remainder of the formulae. This speeds up the evaluation of compatibility of Parikh image abstractions. If resolving satisfiability of formulae takes too long, we can introduce a timeout for SMT solver to stop the computation and not prune the product state.

We also consider combinations of our abstractions, particularly we experiment with computing cheap length abstraction first and computing the Parikh images only when length abstraction fails to prune product states.

Further, we use mintermization for intersection of finite automata as a different approach to processing the initial automata before applying other optimizations. We compute minterms, which can be used instead of transition symbols while retaining all information about the automata to compute Parikh images and other optimization abstractions faster.

We implement the proposed abstractions and evaluate their impact on the emptiness problem and the product construction experimentally. We experiment with a benchmark containing a set of different finite automata obtained from runs of regular model checking tool on verification of pointer program and parametric protocols created in [3] based on method of abstract regular model checking from [4]. Generate products of various combinations of these finite automata. And test emptiness of product languages to solve emptiness problem of their intersections. We focus on the number of trimmed product states and their nature, their position in the product or other significant properties. For certain types of automata, our optimizations work really well. Parikh image abstraction usually trims vast state spaces where length abstraction cannot prune everything and basic product state space explodes (e.g., from 20000 to 10 product states). In addition, the optimizations are sometimes successful at immediately stopping product generation on the first initial product state if the intersection is empty, while, in some cases, product construction would take hours (in one case, 7 hours, compared to 1 minute with Parikh image abstraction).

The contribution of this work can be summarized as follows:

1. heuristics trimming the generated state space of finite automata operations based on abstractions of specific state languages: length abstraction and Parikh image computation; or general approaches as mintermization,

2. optimizations for explored state language abstractions:

    - skipping evaluation of state language abstractions for some product states for sequences of product states in long lines,

3. optimizations specific for length abstraction:

    - generating single lasso automaton for whole finite automaton,
    - efficient evaluation of length abstraction without SMT solver,

4. optimizations specific for Parikh image abstraction:

    - reduced Parikh image to resolve satisfiability of Parikh image formulae faster,

  - resolving Parikh images with incremental SMT solving,
  - resolving Parikh images with timeout for SMT solver,

5. combination of state language abstractions to optimize automata problems, and

6. implementation and experimental evaluation of said heuristics and their optimizations.

Chapter 2

Preliminaries

Let us clarify a few definitions and terms often used throughout this paper. The following definitions are mostly adapted from [11] or [27].

Alphabet is a finite, non-empty set denoted by Σ. Elements of an alphabet are called *symbols* or *letters*. A finite, possibly empty, sequence of symbols over an alphabet is a *word* w from the set of all words Σ^* over an alphabet Σ.

Definition 2.0.1 (Deterministic finite automaton)
A deterministic finite automaton (DFA) is a 5-tuple $A = (Q, \Sigma, \delta, I, F)$, where:

- *Q is a non-empty **set of states**,*
- *Σ is an **input alphabet**,*
- *δ is a **transition function**: $Q \times \Sigma \to Q$,*
- *$I \in Q$ is an **initial state**, and*
- *$F \subseteq Q$ is a **set of final (accepting) states**.*

A *run* of A on input $a_0 a_1 a_2 \dots a_{n-1}$ is a sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_n$, such that $q_i \in Q$ for $0 \le i \le n$, $q_0 = I$ and $\delta(q_i, a_i) = q_{i+1}$ for $0 \le i \le n-1$. A run is *accepting* if $q_n \in F$. A *accepts* a word $w \in \Sigma^*$ if A has an accepting run on input w. A *language* recognized by A is a set $L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$. A single *transition* from δ is denoted as $q \xrightarrow{a} q'$ if $q' \in \delta(q, a)$ and means *one can get from state q to state q' with a transition symbol a*. For every state, DFA has at most one transition for a given symbol. Consequently, DFA has exactly one run on a given word from initial state to one of the accepting states (or non-terminating states[1] in case the word is not accepted by the automaton at all).

Definition 2.0.2 (Non-deterministic finite automaton)
A non-deterministic finite automaton (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, I, F)$, where Q, Σ and F are as for DFA and:

- *δ is a **transition relation**: $\delta : Q \times \Sigma \to P(Q)$, where $P(Q) = \{R \mid R \subseteq Q\}$ is a set of subsets of Q, and*
- *$I = \{q \mid q \in Q\}$ is a non-empty **set of initial states**.*

[1]No accepting state is accessible from them.

For every state and its transition symbol, $P(Q) \in \delta(q, a)$ is a singleton. For example, $\delta(q_1, a) = \{q_1, q_2\}$.

Two finite automata A_1 and A_2 are said to be *equivalent* when both accept the same language: $L(A_1) = L(A_2)$.

For every NFA A exists a corresponding equivalent DFA A'. *Determinization* is a process of converting such NFA to DFA.

Definition 2.0.3 (Powerset (subset) construction)
The powerset construction is a method for creating a corresponding deterministic finite automaton from its equivalent non-deterministic finite automaton. Produces finite automaton A', *where* $Q' = 2^Q$, $F' = \{S \in Q' | S \cap F \neq \varnothing\}$, $I' = I$ *and for* $S \in Q' : \delta'(S, a) = \bigcup_{s \in S} \delta(s, a)$.

Definition 2.0.4 (Product construction)
Given two NFAs $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ *and* $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$ *over the same alphabet* Σ, *operations on* A_1 *and* A_2 *yield a result—a product* A *as a 5-tuple deterministic finite automaton* $A = (Q, \Sigma, \delta, I, F)$ *where:*

- $Q = Q_1 \times Q_2$,
- $\delta : Q \times \Sigma \to P(Q)$,
- $I = I_1 \times I_2$, *and*
- $F = F_1 \times F_2$.

δ is described as $([q_1, q_2], a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$. For pairs of states q_1 and q_2 from A_1 and A_2, respectively, and a common transition symbol a of transitions $q_1' \in \delta_1(q_1, a)$ and $q_2' \in \delta_2(q_2, a)$, we denote a single product transition as $[q_1, q_2] \xrightarrow{a} [q_1', q_2']$, where $[q_1', q_2'] \in \delta([q_1, q_2], a)$ for the corresponding states $[q_1, q_2]$ and $[q_1', q_2']$ in A are called *product states.*

Focusing on *intersection* of automata, the product construction tells that $L(A) = L(A_1) \cap L(A_2)$. Finally, we test the *emptiness* of the intersection language: $L(A)$ does not accept any words.

We work with a basic product construction in Algorithm 1.

Definition 2.0.5 (Galois Connection)
Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ *such that:*

- $\mathcal{P} = \langle P, \leq \rangle$ *and* $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ *are* partially ordered sets *(posets) and*
- *abstraction function* $\alpha : P \to Q$ *and concretization function* $\gamma : Q \to P$ *inverse to* α. $\forall p \in P$ *and* $\forall q \in Q$*:*
$$p \leq \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q.$$

In the terminology of abstract interpretation, P is a *concrete domain* and Q is an *abstract domain.* If α and γ functions form a Galois connection, $\forall p \in P\big(p \leq \gamma(\alpha(p))\big)$. That is, the abstraction may only over-approximate the concrete semantics.

Input : NFA $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$, NFA $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$
Output: NFA $A = (A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

$Q, \delta, F \leftarrow \varnothing$
$I \leftarrow I_1 \times I_2$
$W \leftarrow I$
while $W \neq \varnothing$ **do**
  pick $[q_1, q_2]$ **from** W
  add $[q_1, q_2]$ **to** Q
  if $q_1 \in F_1$ *and* $q_2 \in F_2$ **then**
    add $[q_1, q_2]$ **to** F
  forall $a \in \Sigma$ **do**
    forall $q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)$ **do**
      if $[q_1', q_2'] \notin Q$ **then**
        add $[q_1', q_2']$ **to** W
      add $[q_1', q_2']$ **to** $\delta([q_1, q_2], a)$

Algorithm 1: Classic naive product construction used by our optimization methods to optimize the generated product state space by deciding the compatibility of state language abstractions.

Chapter 3

State Language Abstractions

In this chapter, we introduce several state language abstractions, introduced on product construction and deciding the emtiness problem.

When constructing a product, a considerate number of product states are non-terminating and thus unnecessary. Moreover, the whole product must be constructed before we can determine whether the automata intersection is empty. We want to minimize the number of generated product states when resolving the product construction of automata intersection and deciding the emptiness of intersection language.

We try to guess which product states do not lead to any final states and consequently can be omitted, and the following states do not need to be generated at all. Our optimizations decide the emptiness of parts of the product (or the whole product) already in the process of generating the product (on the fly). We can thus prune non-terminating states before they are added to the product and omit extensive product state space before even considering it in the classic product construction. We achieve this by computing state language abstractions for each state the generated product state consists of and deciding the compatibility of these abstractions.

Our product construction optimizations are applicable on two and more automata, but for the ease of explanation, we consider only two automata. In the following, we will define an abstraction of languages of the states q, $\alpha(q)$. We define two kinds of abstractions: length abstraction $\alpha^{LA}(q)$ and Parikh image abstraction $\alpha^{PI}(q)$. These abstractions represent formulae in first-order predicate logic. Both our $\alpha^{LA}(q)$ and $\alpha^{PI}(q)$ together with their inverse functions form a Galois connection. Hence, they are an over-approximation of state language of q.

For a product state $p = [q_1, q_2]$, we use abstractions of languages of states αq_1 and $\alpha(q_2)$ to quickly detect whether p has an empty language. That can be achieved by checking whether $\alpha(q_1)$ and $\alpha(q_2)$ are compatible. If they are incompatible, product language is empty and p can be pruned (there is no run from p to any final state). Therefore, the optimized product language is the same as the unoptimized product language.

3.1 Length Abstraction of State Languages

In this section, we discuss $\alpha^{LA}(q)$. Length abstraction looks at lengths of words accepted by the state language, creating a set of accepted length. In the following, we first discuss the basic principle of length abstraction. Later, we propose efficiency optimizations for length abstraction. To start with, we introduce lasso automata as an finite automata representation of length abstraction.

3.1.1 Length Abstraction Represented by Lasso Automata

$\alpha^{LA}(q)$ over-approximates the language of q by considering only the accepted lengths of words. This is, if a length of a word does not belong to the length abstraction of q, it cannot be accepted by state language of q, either.

Computing length abstraction over the languages of finite automata states is accomplished using lasso automata (LSA, handle and loop automata)—deterministic finite automata with a unary alphabet (similar as in [1]). They consist of a *handle* (a sequence of states from the initial state) and a *loop* (resolving the cycles in the original automaton) resembling a lasso with a few final states representing the accepted word lengths.

You can create lasso automaton for a state q, $lsa(q)$, by taking the finite automaton A, $q \in Q_A$, setting $I_A = q$, considering all transition symbols as a single transition symbol and determinizing the result with subset construction. $lsa(q)$ is an automaton accepting every length of any word in state language of q. Consequently, it is easy to compute semi-linear set (formulae in the form of a disjunction of linear equations) for the accepted lengths of words, which can be efficiently evaluated. We are computing length formulae for individual states in the product states, checking their satisfiability and constructing only those product states for which the length abstraction formulae are satisfiable.

The length abstraction formulae are generated from $lsa(q)$. For every state q, we get one or more existentially quantified formulae φ in Presburger arithmetic describing language abstracting $\alpha^{LA}(q)$ in the form

$$\varphi : \exists k(|w| = h + l \cdot k)$$

where $|w|$ is a length of a recognized word, h is the length of a handle to a certain final state f, and l is the length of a loop to return to f going through the loop. k is the number of cycles through the loop states until a word ends in f. When multiple depicted formulae are created (because there are more final states or different accepting runs for a single final state in LSA resulting in multiple accepted lengths), we append these formulae with *logical or*:

$$\alpha^{LA} : \exists k(\varphi_1 \vee \ldots \vee \varphi_n)$$

where n is a number of generated φ.

Running Example We demonstrate the construction of $\alpha^{LA}(q_0)$ for initial state q_0 of the following NFA $A_1 = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{0, 1\}, \delta_1, \{q_0\}, \{q_4\})$ where transition relation δ_1 is depicted in Figure 3.1. That is, we construct $LSA(A_1)$. We will use A_1 throughout the section.

A_1 is a non-deterministic finite automaton (see state q_1) and uses multiple input symbols. Due to the fact we work with only recognized word lengths, we can substitute the automaton alphabet with a unary alphabet of a single input symbol $*$[1].

Then, we can generate $lsa(q_0)$, which is its deterministic equivalent. For the final $lsa(q_0)$, generated from A_1' by its product construction determinization, see Figure 3.3. $lsa(q_0)$ accepts any words of lengths of words recognized by state language of q_0.

[1] Even though we do not actually need any particular input symbol, we use * here as an example to depict the process. In general, all we need to know is that there is a transition between two states. The specific transition symbols are not significant for our optimization algorithm.

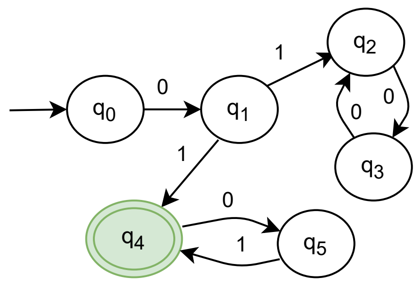

Figure 3.1: Non-deterministic finite automaton A_1.

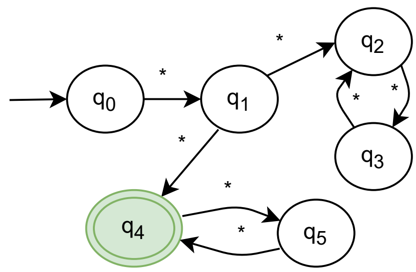

Figure 3.2: Non-deterministic finite automaton A'_1: A_1 with unified transition symbols.

3.1.2 Single Lasso Automaton for Each Original Automaton

When we are constructing a product of finite automata A and B, we do not want to regenerate $lsa(q_A)$ and $lsa(q_B)$ for each product state $p = [q_A, q_B]$. This is inefficient. Due to the nature of LSAs, the successive product states $p' = [q'_A, q'_B]$ generate LSAs very similar to the LSAs for p. We can construct one summary lasso automaton for the whole finite automaton A, $LSA(A)$, which contains all the lasso automata $lsa(q_A)$ for all states in A. Similarly for B.

$LSA(A)$ is a lasso automaton created as a union of all $lsa(q_A)$ for each $q_A \in Q_A$. $LSA(A)$ is a union of all lasso automata states, transitions, final and inital states for each q_A. As a result, α^{LA} for product construction generates only one LSA for each finite automaton (possibly with multiple loops and/or multiple handles). All singleton states in $LSA(A)$ can be initial states. The initial state changes to which state q we want $lsa(q)$ for.

To generate only the necessary $lsa(q)$, we can construct $LSA(A)$ gradually, state by state, for only the currently required q. We generate $lsa(q)$ as the first part of $LSA(A)$. If we need $lsa(q')$ later, we extend $lsa(q)$ with $lsa(q')$ by union of both LSAs. When the new LSA state l is not already present in $LSA(A)$, we add l to $LSA(A)$ and continue with the following states l' until we either create an entirely new loop in $LSA(A)$ or generate l' already in $LSA(A)$ (we can stop generating l' as from now on, all l' are already in $LSA(A)$.

Running Example For our finite automatan A_1, Figure 3.4 shows $LSA(()A_1)$. The initial state is irrelevant as it changes to which state we are currently using.

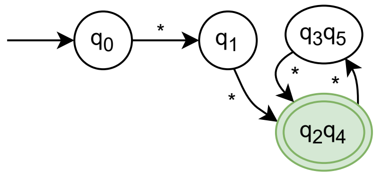

Figure 3.3: Lasso automaton $lsa(q_0)$ for the original NFA A_1 generated from A'_1 by its determination.

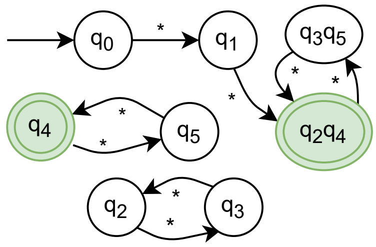

Figure 3.4: Summary lasso automaton $LSA(A_1)$.

3.1.3 Product Construction with Length Abstraction

The core of the product construction remains unchanged, but there are a few differences. The Algorithm 2 shows how we alternate the original product construction to optimize the algorithm with length abstraction.

We call W from line 3 a work set. It stores the potential product states prepared for processing, which we pick from W one by one[2].

The optimization process starts when we pick a product state p from W. Instead of immediately generating new successive product states p', we generate $lsa(q_1)$ and $lsa(q_2)$ to gain length formula. We test the satisfiability of this formula: $sat(\Phi^{LA}(p))$ where

$$\Phi^{LA}(p) : \alpha^{LA}(q_1) \wedge \alpha^{LA}(q_2) \text{ and}$$

$sat(\psi)$ is $True$ iff ψ is satisfiable (Φ **is** sat), $False$ otherwise. On line 9, we check whether $sat(\Phi^{LA}(p))$ holds and store a result as a boolean value to res. We are only interested in the satisfiability test result because we do not need any additional information from the computed formulae. Therefore, a simple boolean value is sufficient. The α^{LA} compatibility check $sat(\Phi^{LA}(p)) = \alpha^{LA}(q_1) \wedge \alpha^{LA}(q_2)$ **is** sat can be implemented in SMT solver as in Algorithm 3:

If $sat(\Phi^{LA}(p))$, i.e., there will be an accepting run using p (see line 10), we add p to Q, possibly to F and generate p'. $\Phi^{LA}(p)$ is passed to an SMT solver to solve its satisfiability. SMT solver returns sat when satisfiable (res is set to $True$) and $unsat$ when unsatisfiable (res is set to $False$). If $unsat$ is returned, length abstractions are incompatible. We have now pruned the generated state space by omitting the product state p.

A note of caution. It is important to understand that we are working only with possible word lengths and when we test the emptiness of intersection of automata, we can resolve

[2] In spite of the fact more approaches are valid, we strongly recommend picking the last added product state from W (see line 7)—using depth-first search—as this allows us to quickly advance through the automaton and get to any final state faster—in case we just want to know whether automata have a non-empty intersection, this change will get us the answer most of the time in less steps. It works even better when implemented with a satisfiable state skipping optimization, explained in Section 3.1.4.

Input : NFA $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$, NFA $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$
Output: NFA $P = (A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(P) = L(A_1) \cap L(A_2)$

```
Q, δ, F ← ∅
I ← I_1 × I_2
W ← I
res ← False
solved ← ∅
while W ≠ ∅ do
    picklast [q_1, q_2] from W
    add [q_1, q_2] to solved
    res ← α^LA(q_1) ∧ α^LA(q_2) is sat
    if res = True then
        add [q_1, q_2] to Q
        if q_1 ∈ F_1 and q_2 ∈ F_2 then
            add [q_1, q_2] to F
        forall a ∈ Σ do
            forall q'_1 ∈ δ_1(q_1, a), q'_2 ∈ δ_2(q_2, a) do
                if [q'_1, q'_2] ∉ solved and [q'_1, q'_2] ∉ W then
                    add [q'_1, q'_2] to W
                add [q'_1, q'_2] to δ([q_1, q_2], a)
```

Algorithm 2: Product construction with length abstraction.

```
smtInit()
smtAdd(k ≥ 0, m ≥ 0)
for φ_q1 ∈ α^LA(q_1) do
    for φ_q2 ∈ α^LA(q_2) do
        smtPush()
        smtAdd(φ_q1.handle + φ_q1.lasso * k = φ_q2.handle + φ_q2.lasso * m)
        res ← smtCheck()
        if res = True then
            break
        smtPop()
```

Algorithm 3: Check length abstraction compatibility with SMT solver.

only such intersections that words lengths are not accepted by both automata. When the test shows there could be some words of certain length accepted by both automata and for that reason by their intersection too—$sat(\Phi^{LA}(p))$—we cannot be sure there truly are any words accepted by both automata with their intersection non-empty, because there may be words of the suggested length, but it may be a different word for each automaton (which differ from one another in the containing symbols or their position in the word). For resolving such cases, we have to proceed with the classic algorithm steps to produce product states according to their original transition symbols, not only by comparing the possible words lengths. With certainty, we can omit only the cases where $\neg sat(\Phi^{LA}(p))$.

Running Example We continue with our running example. The second automaton we will be working with is a NFA $A_2 = (\{s_0, s_1, s_2, s_3\}, \{0, 1\}, \delta_2, \{s_0\}, \{s_3\})$ where δ_2 is depicted in Figure 3.5.

In Figure 3.6, there is $LSA(A_2)$, which we will be using together with $LSA(A_1)$ shown in Figure 3.3 for product construction of A_1 and A_2.

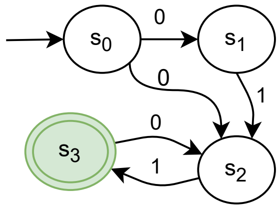

Figure 3.5: Non-deterministic finite automaton A_2.

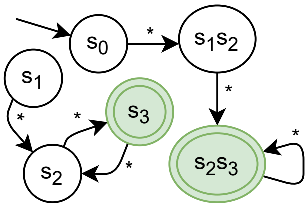

Figure 3.6: Lasso automaton $LSA(A_2)$ for A_2.

When we start the algorithm, we get the following length abstraction formulae for $p = [q_0, s_0]$. From $LSA(A_1)$ for q_0 (q_0 is a new initial state of $LSA(A_1)$), we get an existential formula representing length abstraction $\alpha^{LA}(q_1)$[3]. From $LSA(A_2)$ for s_0 (s_0 as a new initial state of $LSA(A_2)$), we get a formula for length abstraction $\alpha^{LA}(s_0)$[4].

$$\alpha^{LA}(q_0) : \exists k(|w| = 2 \lor |w| = 4 + 2 \cdot k)$$
$$\alpha^{LA}(s_0) : \exists m(|w| = 2 + 1 \cdot m)$$

When we compare $\alpha^{LA}(q_0)$ and $\alpha^{LA}(s_0)$, we get:

$$\alpha^{LA}(q_0) \land \alpha^{LA}(s_0) : \exists k(|w| = 2 \lor |w| = 4 + 2 \cdot k) \land \exists m(|w| = 2 + 1 \cdot m)$$

or in a simplified notation:

$$\alpha^{LA}(q_0) \land \alpha^{LA}(s_0) : \exists k \exists m(2 \lor 4 + 2 \cdot k = 2 + 1 \cdot m).$$

To solve satisfiability of $\Phi^{LA}([q_0, s_0])$, we try to find values of k and m such that $|w|$ equal (some expressions on the left and on the right side of the equation are equal).

In Figure 3.7, we can see the product of A_1 and A_2 being constructed using length abstraction optimization. Red states represent product states whose formulae are resolved as unsatisfiable and therefore the algorithm omits any successive product states—dashed states (such as $q_4 s_2$ or $q_3 s_2$) which are generated in the unoptimized product construction. The green state represents final states in both automata. Here, we have found a solution accepted by both A_1 and A_2. If we desire to resolve only the emptiness problem, we can stop the execution of the algorithm here as we have found one final state—automata have non-empty intersection. The blue state is a normal product state whose significance will be explained in section 3.1.4.

As you can notice in Figure 3.8, the product generated by our algorithm has only 4 product states in comparison to 9 product states generated by the classic algorithm.

[3]This formula consists of two independent disjuncts φ_1 and φ_2 describing there are more possible lengths for accepted words from the same initial state (leading to two independent final states in the automaton).

[4]We are using variable m here instead of k to emphasize variables from different formulae are not dependent on each other—they belong to different LSAs.

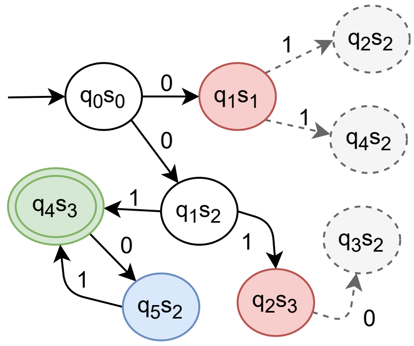

Figure 3.7: Constructed product with depiction of length abstraction optimization.

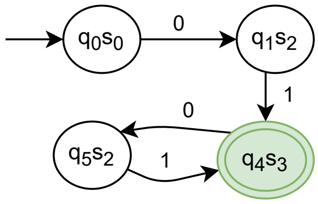

Figure 3.8: Final product minimized by length abstraction.

3.1.4 Optimization with Skipping Satisfiable States

When we take new p from W and check whether $sat(\Phi^{LA}(p))$, it is time to add all the possible successive product states p' to W. When p generates only a single p', we can say with certainty that $sat(\Phi^{LA}(p'))$ as there is only a single branch in the automaton leading from p to a final state (through p'). p' is *skippable*, iff there exists p for which $sat(\Phi^{LA}(p))$ and whose only successor is p'. We add p' to W with the information of being skippable. If p' is already in W, we append the information to p' in W.

We skip checking for $sat(\Phi^{LA}(p'))$ when we pick p' from W. We can immediately check for final states and generate the successive product states. This optimization saves us generating the length abstraction formulae for p' and testing the formulae in the SMT solver for their satisfiability and even possibly reducing the number of states generated for both initial automata.

If automata have long lines (with non-splitting branches), this will prove extremely useful, because only a few proper iterations with formulae computing and executing SMT solver will be executed. The application of skipping satisfiable states is depicted in Algorithm 4. The line 9 from Algorithm 2 is substituted with the contents of Algorithm 4.

if `skippable`$([q_1, q_2])$ **then**
    $res \leftarrow True$
else
    $res \leftarrow \alpha^{LA}(q_1) \wedge \alpha^{LA}(q_2)$ **is** sat

Algorithm 4: Substitution of line 9 in Algorithm 2 with skipping satisfiable states.

The only change is a test for every checked p, which decides whether p can be skipped. You can see that we proceed with SMT solver satisfiability check only for p which are generated from the product states with multiple transitions generating p and at least one more product state (in general at least two new potential product states). If only one p was generated earlier from a product state with satisfiable formulae of non-zero length[5], we skip the check for $sat(\Phi^{LA}(p))$ and continue to generating its successive states immediately.

You can notice there is one skippable state in the former example, which had to be evaluated and tested for satisfiability earlier. The blue state in Figure 3.7 is such a skippable state. In our case for state q_5s_2, when only one new state is generated from state q_4s_3 while this state is resolved as satisfiable (with non-zero length—otherwise, if q_5s_2 did not lead back to q_4s_3, q_5s_2 would be skippable even though it would not lead to any final state), newly generated product state has to be satisfiable as well, because the check for q_4s_3 already considered the state q_5s_2 as its only way to any final state with non-zero length.

When we have a series of such states, we can highly optimize generating the whole branch with only one initial check for satisfiability. In real world examples, there are often automata with long branches splitting into multiple branches only occasionally. We will check for satisfiability for all the initial states of each new branch and then either omit the entire branch (if *unsat* is returned) or skip checking satisfiability in the entire branch (if *sat* is returned).

3.1.5 Resolving Length Abstraction Satisfiability without SMT Solver

Evaluating satisfiablity of length abstractions formulae in SMT solver is expensive. We try to replace SMT solver with a specialized structure which transforms the problem of solving satisfiability of length abstraction formulae to evaluating satisfiability of a linear congruence equations.

Length abstraction formulae have the same, simple structure. Length abstraction can be implemented as a set of length formulae represented as a two-tuple of handle length and lasso length. Therefore, we can easily compare such sets in order to resolve satisfiability of length abstraction formule wihout SMT solver[6]. $\Phi^{LA}(p)$ forms a set of linear congruence equations, which can be resolved just by utilizing basic mathematical operations and properties of linear congruences.

The Algorithm 5 shows how to determine $sat(\Phi^{LA}(p))$ from line 9 using linear congruences.

1 **for** $\varphi_{q_1} \in \alpha^{LA}(q_1)$ **do**
2   **for** $\varphi_{q_2} \in \alpha^{LA}(q_2)$ **do**
3   **if** $\varphi_{q_1}.handle = \varphi_{q_2}.handle$ **then**
4             $res \leftarrow True$
5   **else if** $\varphi_{q_1}.handle > \varphi_{q_2}.handle$ **then**
6             $res \leftarrow$ `solveForOneHandleLonger`$(\varphi_{q_1}, \varphi_{q_2})$
7   **else**
8             $res \leftarrow$ `solveForOneHandleLonger`$(\varphi_{q_2}, \varphi_{q_1})$

9 $res \leftarrow False$

Algorithm 5: Check satisfiability using length abstraction algorithm without SMT solver.

We execute the following steps for each equation

$$\varphi_{q_1}.handle + \varphi_{q_1}.lasso \cdot k = \varphi_{q_2}.handle + \varphi_{q_2}.lasso \cdot m.$$

[5]If the satisfiable length is zero, we are in a final product state and p generated from final state might not lead to any final state, but the length abstractions for final state are compatible, of course.

[6]Note that, as with SMT solver, we do not need to find specific values for which the formulae are satisfiable. We are only interested in deciding whether such values exist.

If the handle lengths of φ_{q_1} and φ_{q_2} are equal, there are words of the same length accepted by both $\alpha^{LA}(q_1)$ and $\alpha^{LA}(q_2)$ (they are mutually compatible) without stepping into the loops of $lsa(A_1)$ and $lsa(A_2)$. Otherwise, handle lengths differ, and we must consider lengths of loops in our determination of compatibility of $\alpha^{LA}(q_1)$ and $\alpha^{LA}(q_2)$.

We now have to determine $sat(\varphi_{q_1} \wedge \varphi_{q_2})$ for abstractions with one handle longer. This is solved by the function `solveForOneHandleLonger` in Algorithm 6.

```
1  Function solveForOneHandleLonger(φ_l, φ_s):
       Data: Input length abstraction formulae of potential product state.
       φ_l: Length abstraction formula with the longer handle,
       φ_s: Length abstraction formula with the shorter handle.
       Result: bool: True if satisfiable, False otherwise.
2      φ_l.handle ← φ_l.handle − φ_s.handle
3      φ_s.handle ← 0
4      if φ_l.lasso = 0 and φ_s.lasso = 0 then
5          return False
6      else if φ_s.lasso = 0 then
7          return False
8      else if φ_l.lasso = 0 then
9          it ← 0 // Current length resembling the iteration of the shorter lasso loop.
10         while it ≤ φ_l.handle do
11             if it = φ_l.handle then
12                 return True
13             else
14                 it ← it + φ_s.lasso
15         return False
16     else
17         gcd ← getGCD(φ_l.lasso, φ_s.lasso)
18         if gcd = 1 then
19             return True
20         else
21             y ← −φ_l.handle
22             while y < gcd do
23                 y ← y + φ_s.lasso
24             if (y mod gcd) = 0 then
25                 return True
26             else
27                 return False
```

Algorithm 6: Solve satisfiability of length abstraction formulae for one handle longer.

First, to simplify the equation (line 2), we move handle lengths from the side of the equation with the shorter handle φ_s to the side with the longer handle φ_l to solve:

$$\varphi_l.handle + \varphi_l.lasso \cdot k = \varphi_s.lasso \cdot m \tag{3.1}$$

which represents the number of loops a word must make in φ_s to be accepted by φ_l (as if with shorter $\varphi_l.handle$).

If both φ_l and φ_s have no loops (line 4), $\varphi_{q_1} \wedge \varphi_{q_2}$ is unsatisfiable because the handles differ. Else, if only φ_s has no loop (line 6), every word accepted by φ_s is shorter than words accepted by φ_l and the formulae cannot be satisfiable.

Else, if only φ_l has no loop (line 8), we can try to manually iterate over loops in φ_s to see whether the word length difference between handles can be equalized by looping in $\varphi_s.lasso$.

Otherwise, both φ_l and φ_s have loops (line 16). We can apply linear congruence properties to the equation 3.1 to determine whether the formulae are satisfiable. The equation 3.1 says that if formulae are satisfiable, the left side of the equation is divisible by some multiple of $\varphi_s.lasso$. We can rewrite that in a linear congruence as follows:

$$\varphi_l.handle + \varphi_l.lasso \cdot k \equiv 0 \pmod{\varphi_s.lasso} \tag{3.2}$$

$$\varphi_l.lasso \cdot k \equiv -\varphi_l.handle \pmod{\varphi_s.lasso} \tag{3.3}$$

which is the same as solving a linear Diophantine equation

$$\varphi_l.lasso \cdot k - \varphi_s.lasso \cdot m = -\varphi_l.handle. \tag{3.4}$$

Qualities of multiplicative inverse [8, 17], based on Bézout's identity [8], say that iff $\varphi_l.lasso$ and $\varphi_s.lasso$ are relatively prime (coprime)—the greatest common divisor (GCD) of $\varphi_l.lasso$ and $\varphi_s.lasso$ is equal to 1—there exists a multiplicative inverse for $\varphi_l.lasso$ in modulo $\varphi_s.lasso$ which ensures that linear congruence 3.3 is always solvable for some $\varphi_l.lasso$ in modulo $\varphi_s.lasso$[7]. Therefore, the formulae are satisfiable.

Otherwise, $\varphi_l.lasso$ and $\varphi_s.lasso$ are not coprime (GCD is different from 1) and by properties of linear Diophantine equations [17], iff GCD precisely divides y without a remainder where y is the right side of the linear congruence 3.3 or its any congruent equivalent, there exist solutions to the linear congruence[8]. Otherwise, there are no solutions.

3.2 Parikh Image Abstraction of State Languages

Length abstraction is a simple and fast optimization, but can be too coarse to detect non-terminating states in some cases. In this section, we present an abstraction of state languages with Parikh images, α^{PI}, which aims to replace length abstraction to make the abstraction more precise to prune larger quantities of product state space.

Parikh images provide more information about the finite automata than simple length abstraction. While length abstraction considers only accepted word lengths without knowing which transition symbols are actually in the transitions, Parikh image abstracts accepted words to numbers of occurrences of specific transition symbols in words regardless of their position in said words. Thus, Parikh image abstraction allows us to more precisely determine the emptiness problem. However, Parikh image computation itself is expensive. The question is, whether the added computation time compensates for more precise product generation with higher state pruning capabilities.

We will introduce an algorithm for Parikh image abstraction α^{PI} applied on each product state $p = [q_1, q_2]$ to decide the compatibility of $\alpha^{PI}(q_1)$ and $\alpha^{PI}(q_2)$.

3.2.1 Parikh Image

We derive our Parikh image construction from the Parikh's theorem [23] described in [12], creating a semi-linear Parikh image formulae for the given regular language as a set of Parikh images for each word in the language. However, our usage of Parikh image of some regular language (and therefore of the corresponding finite automaton recognizing such regular language) is restricted to determining the compatibility of Parikh image state

[7]We can get the precise solution by multiplying both sides of the equation with the multiplicative inverse.

[8]We can apply extended Euclidean algorithm to find the precise values for the Diophantine equation 3.4.

language abstractions. Therefore, we only test for satisfiability of Parikh image formulae describing $\alpha^{PI}(q_i)$. We use SMT solver to resolve the satisfiability of Parikh image formulae of the current potential product state.

Given an NFA $A = (Q, \Sigma, \Delta, I, F)$ where I is a singleton $I = \{q_0\}$, Parikh image formula φ (as described in [22] for solving string constraints) consists of several constraints in conjunctive normal form. φ describes runs of A. Each satisfiable assignment defines properties of the run. φ consists of the following conjuncts:

1. Foremost, we define a variable u_q for each state $q \in Q$. u_q defines how many times we enter q and exit q by specifying the difference between the number of entries and exits. We construct equations with u_q for a run as follows:
   - $u_q = 1$ for $q \in I$,
   - $u_q \in \{0, -1\}$ for $q \in F$ and
   - $u_q = 0$ for $q \in Q \setminus (I \cup F)$.
2. Second, we define a variable y_t for each transition $t \in \Delta$ such that $y_t \geq 0$ describing how many times is t used in the run.
3. We can now present an equation introducing a connection between u_q and y_t to evaluate the difference between the number of entries and exits for each $q \in Q$ as follows:
$$u_q + \sum_{t \in \Delta_q^+} y_t - \sum_{t \in \Delta_q^-} y_t = 0.$$
where Δ_q^+ is a set of ingoing transitions $\Delta_q^+ = \{(q', a, q) \in \Delta\}$ and Δ_q^- is a set of outgoing transitions $\Delta_q^- = \{(q, a, q') \in \Delta\}$ from the given state q.
4. Furthermore, we need to make sure that the states used in runs described by the satisfying assignemnts are connected and start in the initial state. Variable z_q for each $q \in Q$ is introduced. z_q represents the length of any path from I to q in a spanning tree of the subgraph with $y_t \geq 0$. If $z_q = 0$, there is no path from I to q and the state q is not used in the run. $z_q > 0$ means there is a path feom I to q and q is used in the run.

   If $q \in I$, we add a constraint $z_q = 1 \wedge y_t \geq 0$. Otherwise,
$$(z_q = 0 \wedge \bigwedge_{t \in \Delta_q^+} y_t = 0) \vee \bigvee_{t \in \Delta_q^+} (y_t \geq 0 \wedge z_{q'} \geq 0 \wedge z_q = z_{q'} + 1).$$
   If the distance z_q is 0, q is not in the run.
5. Last but not least, we declare the only free variable $\#_a$ for each transition symbol $a \in \Sigma$. $\#_a$ describes the number of occurrences of a in accepted words regardless of their position in the words (the number of a in the run). $\#_a$ is the only variable common to different Parikh image abstractions when we test their compatibility. The constraint $\#_a = \sum_{t=(q,a,q') \in \Delta} y_t$ ensures $\#_a$ is consistent with the number of used t with a.

We gain an existentially quantified formula φ in Presburger arithmetic describing language abstracting α^{PI} for A with free variables $\#_a$:
$$\alpha^{PI} : \exists u_{q_1}, \ldots, u_{q_n}, z_{q_1}, \ldots, z_{q_n}, y_{t_1}, \ldots, y_{t_m} (\varphi)$$

where $n = |Q|$ is the number of states and $m = |\Delta|$ is the number of transitions in the finite automaton.

Notice that α^{PI} is an existential formula, which is great for SMT solving where computing with universal quantifiers can take a long time. Smt solver are specialized on efficient solving of existential or quantifier-free formulae.

As for length abstraction for product state $p = [q_1, q_2]$, we decide compatibility of Parikh image formulae $\alpha^{PI}(q_1)$ and $\alpha^{PI}(q_2)$ as follows: $sat(\Phi^{PI}(p))$ such that

$$\Phi^{PI}(p) : \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)^{9}.$$

3.2.2 Product Construction with Parikh Image Abstraction

We introduce the basic product construction using Parikh image abstraction. The algorithm is analogous to the product construction algorithm optimized by length abstraction from Algorithm 2. The difference is that we now compute Parikh image formulae and determine their satisfiability instead of generating lasso automata and determining satisfiability of length abstraction formulae.

We use Parikh image formulae to determine whether p is to be added to the product P. As for length abstraction, we test whether Parikh image abstractions are compatible (a conjunction of Parikh image formulae is satisfiable). Therefore, instead of length abstraction on line 9 in Algorithm 2, we compute Parikh image abstractions: line 9 is replaced with

$$res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2) \textbf{ is } sat$$

We can see our proposed algorithm using Parikh image computation to optimize product construction in the Algorithm 7. Parikh image formulae are computed on line 9 and their satisfiability is determined.

3.2.3 Reduced Parikh Image

The presented Parikh image would work well regarding its pruning capabilities. However, the described Parikh image computation requires extensive resources and computation time and we need Parikh images computed only for determining the emptiness of the intersection. Given that most of the computation time is spent by the evaluation of Parikh image conjuncts in SMT solver, we want to minimize the number of Parikh image conjuncts SMT solver needs to evaluate for each φ.

Consequently, we infer our reduced Parikh image from the shown Parikh image to further optimize Parikh image computation. We modify several conjuncts in Parikh image formula and unify initial states and accepting states to simplify the formula and reduce its complexity.

Due to how we have reduced our Parikh image, we work only with finite automata with a single initial state and a single accepting state. However, we can easily convert any finite automaton into the required format with adding two new states: one for a new initial state from which one can transition to all previous initial states and one for a new accepting state to which lead all previous accepting states. The previous initial and accepting states are changed to common automata states.

Our reduced Parikh image consists of the following conjuncts:

[9] In reference implementation, we replace existential formulae with quantifier-free formulae with renamed variables without existential quantifiers.

Input : NFA $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$, NFA $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$
Output: NFA $P = (A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(P) = L(A_1) \cap L(A_2)$

$Q, \delta, F \leftarrow \varnothing$
$I \leftarrow I_1 \times I_2$
$W \leftarrow I$
$res \leftarrow False$
$solved \leftarrow \varnothing$
while $W \neq \varnothing$ **do**
  picklast $[q_1, q_2]$ **from** W
  add $[q_1, q_2]$ **to** $solved$
  $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat
  if $res = True$ **then**
    add $[q_1, q_2]$ **to** Q
    if $q_1 \in F_1$ ***and*** $q_2 \in F_2$ **then**
      add $[q_1, q_2]$ **to** F
    forall $a \in \Sigma$ **do**
      forall $q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)$ **do**
        if $[q_1', q_2'] \notin solved$ ***and*** $[q_1', q_2'] \notin W$ **then**
          add $[q_1', q_2']$ **to** W
        add $[q_1', q_2']$ **to** $\delta([q_1, q_2], a)$

Algorithm 7: Product construction with Parikh image abstraction.

1. We use the conjuncts 1, except now we restrict u_q for each final state to have only the value -1, i.e.:
$$u_q = -1 \text{ for each state } q \in F.$$
We can perform this reduction, because we know for sure that by unifying final states of the automaton into one abstract final state, there will be exactly only one final state where all words accepted by the automaton end, but none passes through this state earlier.

2. The conjuncts 2 and 3 remain unchanged, the same holds for conjuncts 5.

3. However, we completely omit the conjuncts for z_q. The reason is that, as we have found out, the difference in pruning capabilities of Parikh image with or without the conjuncts 4 on our benchmark automata is insignificant in comparison to the computation time spared by removing these conjuncts.

   The reason conjuncts 4 are so computationally costly is that they are complex for even simple automata. Even then, if want to keep them, we can include these conjuncts, but, we can reduce their complexity by not having to compute z_q lengths for inital and final states.

   The constraint for when q is an initial state $(z_q = 1 \wedge y_t \geq 0)$ remains unchanged as a starting length for other states. However, for every other state, we remove the possibility of $y_t = 0$ and $z_{q'} = 0$ in the second half of the conjuncts (as the option cannot occur with unified initial and final states). The conjuncts look like this:
$$(z_q = 0 \wedge \bigwedge_{t \in \Delta_q^+} y_t = 0) \vee \bigvee_{t \in \Delta_q^+} (y_t > 0 \wedge z_{q'} > 0 \wedge z_q = z_{q'} + 1).$$

Skippable States Optimization

Same as for the length abstraction, we can make use of skipping satisfiable product states optimization. When $sat(\Phi^{PI}(p))$ for some potential product state $p = [q_1, q_2]$ and p generates only one consecutive potential product state $p' = [q_1', q_2']$ such that $p \xrightarrow{a} p'$ where $a \in \Sigma$, we can skip computing Parikh images for p' as we know for sure $sat(\Phi^{PI}(p'))$ in order to get a satisfiable result for Parikh image for p. We can add this functionality to our previous algorithm by replacing line 9 with the content of Algorithm 8.

if `skippable`$([q_1, q_2])$ **then**
  $res \leftarrow True$
else
  $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat

Algorithm 8: Parikh image computation with skippable states optimization.

3.2.4 Optimization with Incremental SMT Solving

Parikh image formulae are large and SMT solving is expensive. We have to recompute Parikh image formulae for every potential product state. However, formulae generated for different product states in one intersection problem are very similar. Large parts of Parikh image formulae do not change between the product states at all.

We try to use SMT solver with incremental SMT solving to reuse parts of the previous computation in the next one. We can specify parts of the Parikh image formulae in the SMT solver once without passing them to the solver for each product state. Further, once a formula have been computed, the solver can use its cache to reuse parts of the computation[10]. In this section, We explain how we use incremental SMT solving for Parikh images in product construction to compute similar, consecutive Parikh image formulae faster.

Notice that some conjuncts of Parikh image remain unchanged for the whole automaton, i.e., for every product state. Only some conjuncts which work with inital states (conjuncts 1 and 4) have to be rewritten, because the only difference between states in two different product states are the different initial states.

Assume finite automata A_1 and A_2 (whose intersection we generate) and a product state $p = [q, s]$ where $q \in Q_{A_1}, s \in Q_{A_2}$. The changes of conjuncts in φ_{A_1} and φ_{A_2} are caused by moving (setting) the states in both A_1 and A_2 corresponding to p as new initial states $I_{A_1} = \{q\}$ and $I_{A_2} = \{s\}$ as we proceed further into the automata with product construction. We start with the abstract initial states (one for each original automata, $I_{A_1} = \{q_0'\}$ and $I_{A_2} = \{s_0'\}$).

First, we compute $\Phi^{PI}(p_0)$ such that $p_0 = (q_0', s_0')$. Iff $sat(\Phi^{PI}(p_0))$, we generate new potential product states (e.g., $p_1 = (q_1, s_1)$ and $p_2 = (q_1, s_2)$). Now we need to check whether to include p_1 and p_2 to the generated product, i.e., check that $sat(\Phi^{PI}(p_1))$ and $sat(\Phi^{PI}(p_2))$, respectively. Taking p_1, we set new initial states $I_{A_1} = \{q_1\}, I_{A_2} = \{s_1\}$. Similarly, for p_2, we would set $I_{A_1} = \{q_1\}, I_{A_2} = \{s_2\}$.

We now need to change every mention of initial states in φ_{A_1} and φ_{A_2} because the initial states are different from those we used at the start (q_0' and s_0') and for which we already computed $\Phi^{PI}(p_0)$. We now introduce an optimization of Parikh image computation which

[10]Consequently, the computation of the first Parikh image takes longer than for the next states.

precomputes unchanged conjuncts only once and recomputes only conjuncts mentioning initial states.

Persistent and State Specific Clauses

To present optimization with incremental SMT solving, we split $\alpha^{PI}(q)$ conjuncts into two groups: persistent clause and state specific clause.

Persistent clause represents Parikh image conjuncts which can be precomputed once for all states in the finite automaton and used throughout the whole product construction. Persistent clause consists of unchanged conjuncts of reduced Parikh image described in 3.2.1: conjuncts 2, conjuncts 3 and conjuncts 5.

State specific clause consists of conjuncts which change with every product state p, and as such have to be constructed and recomputed for every satisfiability test. The process of recomputing state specific clauses is the most expensive part of the product construction algorithm using Parikh images. Therefore, our goal is to minimize the number of conjuncts in a state specific clause as much as possible. The state specific clause consists of conjuncts 1 in reduced Parikh image as they directly change according to initial states and, optionally, if we want to include z_q conjuncts, conjuncts 3. We would need to recompute z_q conjuncts for each potential product state too because the conjuncts compute with initial states.

It is worth to note that the conjuncts 3 in reduced Parikh image manipulate with initial states but the structure of the conjuncts could be reversed to compute connectedness of the automaton in *reversed* order, from the accepting states to the initial states. In that case, the conjuncts could be reconstructed as a part of the persistent clause dependent on accepting states which remain unchanged (the abstract accepting state) for the entire time. This additional optimization might be worth inspecting. Because the inclusion of conjuncts 3 does not generate smaller state spaces with our benchmark automata, we did not investigate further yet.

Algorithm for Incremental SMT solving Using Parikh Image

To implement incremental SMT solving to our current Parikh image computation shown in Algorithm 7, we need to make the following adjustments.

We need to precompute persistent clauses once for both A_1 and A_2. We insert a new line to our algorithm between lines 5 and 6. The new line contains a call to a function `addPersistentClauses()` which precomputes persistent clauses for both A_1 and A_2. Note that the function is called only once, before we enter the *while* loop for iterating over potential product states.

We compute state specific clauses as normal when we ask whether $sat(\Phi^{PI}(p))$ when we are checking compatibility of both α^{PI} on line 9. However, we push the previously precomputed state persistent clauses to the SMT solver stack. This preserves them when the current state specific clauses are dropped after $sat(\Phi^{PI}(p))$ is resolved. For a pseudocode of the replacement of line 9, see Algorithm 9.

```
smtPush()
res ← α^{PI}(q_1) ∧ α^{PI}(q_2) is sat
smtPop()
```

Algorithm 9: Add state specific clauses to SMT solver for incremental SMT solving optimization.

The line 2 computes Parikh image formulae and determines their satisfiability, as explained in Section 3.2.3.

3.2.5 Optimization with SMT Solver Timeout

In the case of Parikh images computed with SMT solver, it is easier to determine $\neg sat(\Phi^{PI}(p))$ than $sat(\Phi^{PI}(p))$. Based on our experiments, we use timeout functionalities of SMT solver to speed up the process of resolving satisfiability of potential product states.

We define a maximal amount of time SMT solver can compute $sat(\Phi^{PI}(p))$ for a single product state p to resolve its satisfiability. If SMT solver resolves $sat(\Phi^{PI}(p))$ before the time runs out, we proceed as normal. However, if the time runs out, the result of the satisfiability test is unknown and we must presume $\Phi^{PI}(p)$ could be satisfiable: we must set res to $True$.

This approach resolves $sat(\Phi^{PI}(p))$ of an over-abstraction described previously. We prune such potential product states that $sat(\Phi^{PI}(p))$ can be resolved quickly (within the defined timeout) while allowing the inclusion of some potential product states which are in fact unnecessary to the generated product. Nevertheless, we find pruning capabilities of this optimization satisfactory and the computation time decreases noticeably.

The timeout is chosen empirically. One has to experiment with their finite automata. The ideal timeout can vary for different benchamarks. One timeout is usually successfully usable for operations on similar finite automata. The timeout is directly proportional to a precision of Parikh image abstraction and reversely proportional to the scale of Parikh image over-abstraction.

3.3 Combination of State Language Abstractions

Length abstraction is fast but coarse; Parihk image abstraction is precise but expensive. We can combine both abstractions to take advantage of respective strengths of our abstractions. In this section, we present an algorithm which introduces a modification to evaluation of compatibility of state langauge abstractions. We use both length abstraction and Parikh image computation to determine satisfiability of state abstraction to optimize product construction. The pruning capabilities remain the same as if we computed Parikh image alone or even better in casses where Parikh image computation times out.

The Algorithm 10 shows how we apply our modifications on a single evaluation of compatibility of abstractions.

if $\alpha^{LA}(q_1) \wedge \alpha^{LA}(q_2)$ ***is*** unsat **then**
    $res \leftarrow False$
else
    $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat
    if $res = Unknown$ **then**
        $res \leftarrow True$

Algorithm 10: Implementation of checking compatibility of state abstractions using both length abstraction and Parikh image computation optimizations.

First, we test whether α^{LA} alone can prune the generated product state space by omitting the current potential product state $[q_1, q_2]$ if $\neg sat(\Phi^{LA}([q_1, q_2]))$. If length abstraction succeeds in omitting $[q_1, q_2]$ from the product, we do not need to compute Parikh images for $[q_1, q_2]$ and can continue with the product construction as if $\neg\Phi^{PI}([q_1, q_2])$. Otherwise, we continue with Parikh image computation for $[q_1, q_2]$ (resolving satisfiability of its formulae as in the basic Parikh image algorithm from Algorithm 7).

3.4 Abstraction of State Languages with Mintermization

In this section, we introduce a method of optimizing operations on finite automata using minterms [21] for the finite automata. Minterm computation abstracts the state language of automata differently than what we have explored so far, allowing us to follow a diverse set of characteristics about the state language. We can afterwards make use of computed minterms for the automata with other optimization methods introduced in this paper, as well as another optimization approaches.

Foremost, we give an algorithm for minterm computation adapted from [10], further defined and expanded in [20] for simulation algorithms for symbolic automata and now optimized for product construction to compute minterms for the non-empty multiset of input finite automata $A = A_1, A_2, \dots, A_n$ where n equals the number of finite automata, which we desire to execute the required automata operation on. Gained minterms abstract automata state language in such a way we do not lose any information about the former automata (minterms are not an over-approximation of the original automata), but might create a more concise finite automata which will be easier to work with in our other optimization methods and may significantly decrease the computation time required for optimizations such as Parikh image computation.

The general idea is to get sets of transition symbols between two states for all our considered finite automata. Compute minterms from these sets and substitute transition symbols between two states in our automata with corresponding minterms created from these transition symbols.

For now, let us explain what minterms are and how you can generate them.

Definition 3.4.1 (Minterms)
Given an NFA $A = (Q, \Sigma, \delta, I, F)$, let $\Phi = \{\varphi_1, \varphi_2, \dots, \varphi_n\}$ be a finite set of non-empty finite sets of transition symbols $\varphi_i = \{a \mid a \in \Sigma \wedge q \xrightarrow{a} q'\}$ for $1 \le i \le n$ where $q, q' \in Q$, n equals the number of state pairs (q, q') such that $q \xrightarrow{a} q'$ where $q' \in \delta(q, a)$.

We call φ_i a *transition set* for the given pair of automaton states q, q'. We denote Ψ or Minterms(Φ) as a set of all minterms ψ for A such that

$$\Psi = Minterms(\Phi) = \left\{ \psi = \bigcap_{1 \le i \le n} \psi_i \;\middle|\; \forall i \in \{1, \dots, n\}\big((\psi_i \in \{\varphi, Q \setminus \varphi\}) \wedge \psi \neq \emptyset\big)\right\}.$$

Minterms are computed once, at the beginning of the optimization process for all considered finite automata We generate so called *minterm tree* with nodes as intersection between sets of transition symbols in case the intersection is non-empty. Each node can have up to two children, representing intersection with the next transition set and its complement, respectively.

When such minterms for the given automaton are computed, we can abstract the state language of the automaton by replacing transitions from the state by their corresponding minterms. We say minterm ψ is created from the set of transition symbols $\varphi \in \Phi$ if φ is used in the intersection defining ψ in its direct form, not as a complement $Q \setminus \varphi$.

Notice that we can compute minterms over multiple NFAs, which allows us to use minterms state language abstraction for optimization of operations on those automata.

Given finite automata $A_1 = (\{s_0, s_1, s_2, s_3\}, \Sigma, \delta_1, \{s_0\}, \{s_3\})$ and $A_2 = (\{q_0, q_1, q_2\}, \Sigma, \delta_2, \{q_1\}, \{q_0\})$ over alphabet $\Sigma = \{a, b, c, d\}$ with δ_1 and δ_2 according to Figure 3.9 and Figure 3.10, respectively, the Figure 3.14 depicts how we could mark each transition set in our automata to be used in mintermization process. For example, a transition set φ_1 could be a set of transition symbols from state s_0 to s_1: $\varphi_i = \{a, b, d\}$. Similarly, we mark the remaining transition sets. Now, we can proceed to execute mintermization operations.

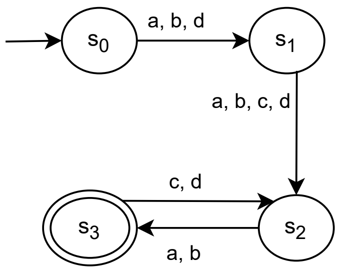

Figure 3.9: Finite automaton A_1 with transitions δ_1.

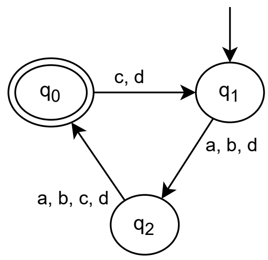

Figure 3.10: Finite automaton A_2 with transitions δ_2.

Figure 3.11: Finite automata A_1 and A_2 used as example automata for mintermization.

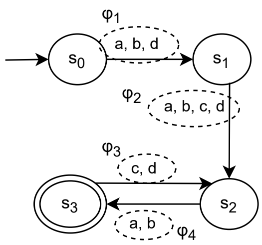

Figure 3.12: Finite automaton A_1 with transition sets φ_i.

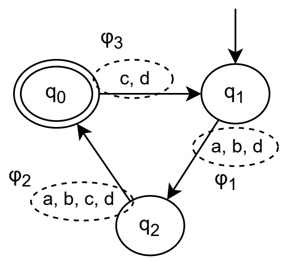

Figure 3.13: Finite automaton A_2 with transition sets φ_i.

Figure 3.14: Finite automata A_1 and A_2 with marked transition sets used in mintermization.

Computation of minterms for A_1 and A_2 is illustrated in Figure 3.15 in a diagram.

We start with the whole alphabet of both automata[11] at the top of the minterm tree to be generated. Afterwards, we iterate over transition sets. For each transition set φ_i, we compute the intersection of the current minterm tree leaves with:

- the current transition set φ_i and store the result as a left node of this particular tree node,
- the complement of the current transition set $Q \setminus \varphi_i$ and store the result as a right tree node of this particular tree node.

[11]If the automata had non-equal alphabets, we would start with their intersection: $\Sigma = \Sigma_1 \cap \Sigma_2$. This is an optimization specific to product construction: If some transition symbols are not used by every finite automaton, we can safely omit such symbols as they are definitely not present in the intersection of these automata.

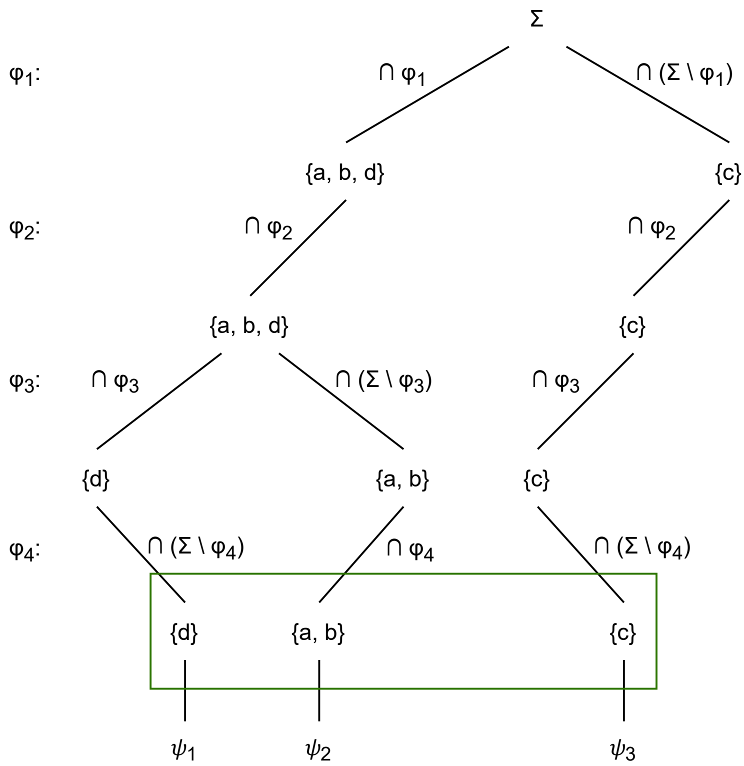

Figure 3.15: Mintermization process executed on example finite automata A_1 and A_2. We start with the whole alphabet and make our way down through all mintermization sets φ_i, where $1 \leq i \leq n$. For each mintermization set, we compute the intersection of the preceding set with the current mintermization set φ_i. The results are shown in the diagram as the nodes of the tree. When operations on all mintermization sets were executed, the leaves of the tree (indicated by the green square) represent the final minterms for the given mintermization sets Φ over the given alphabet Σ. We denote each minterm ψ_i, where $1 \leq i \leq |\Psi|$ where $|\Psi|$ represents the total number of generated minterms.

If the intersection is empty, we omit creating the corresponding child node entirely. In the end, we are left with a complete minterm tree for the given set of transition sets Φ representing the specified finite automata.

The acquired minterms are:

$$\Psi = Minterms(\Phi) = \big\{\{d\}, \{a, b\}, \{c\}\big\} = \{\psi_1, \psi_2, \psi_3\}.$$

We can now substitute the former transition sets φ_i for finite automata with the appropriate minterms $\psi_j, 1 \leq j \leq |\Psi|$ which were created from the specific transition sets $\varphi_i \in \Phi$ such that φ_i is used in its direct form (not as a complement) in the process of computing ψ_j (optimized for product construction). The gained automata can be seen in Figure 3.18.

As we can see, we are able to get rid of some transition symbols and reduce the alphabet as well as the number of transitions in finite automata. Considering we have minterms over alphabet of A, we know that the intersection of two minterms has to be an empty set and that $\forall \psi \in \Psi (\psi \subseteq \varphi, \varphi \in \Phi)$ if ψ is created from φ. Important improvement of using minterms in product construction is the fact that $|\Psi| \leq |\Sigma|$ instead of at most $2^{|\Sigma|}$ as is the case for

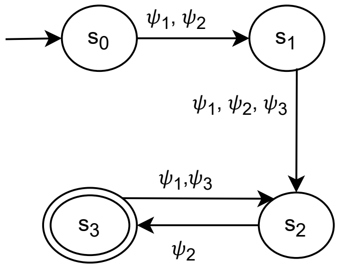

Figure 3.16: Finite automaton A_1 with transitions substituted by corresponding minterms $\psi_i \in \Psi$ created from these transition sets.

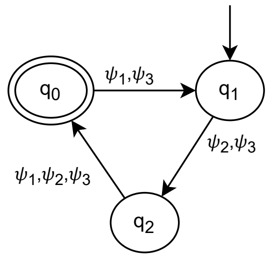

Figure 3.17: Finite automaton A_2 with transitions substituted by corresponding minterms $\psi_i \in \Psi$ created from these transition sets.

Figure 3.18: Finite automata A_1 and A_2 with substituted transitions with minterms in the process of mintermization.

minterms over general predicates for general operation (e.g., [20]). We make use of these points further.

We can use the method of minterm computation with length or Parikh image computation state language abstraction optimizations. We choose this approach in order to mitigate the disadvantages of lower pruning capabilities of length abstraction for some finite automata or demanding Parikh image computation, especially for automata with multitude of transitions between two states varying only in transition symbols, which require considerate time to compute and evaluate.

This method proceeds to represent such sets of transitions between two states with (ideally) only a single minterm representing these transitions. We can therefore apply any previously mentioned optimization methods (or any other known optimization method) on such modified automata with minterms as their transition symbols to construct their product without the need to compute, for example, Parikh image with every single transition symbol between two states. We can now compute possibly fewer transitions with the acquired minterms instead. Worst case is that the minterms do not reduce any transition symbols and we continue with the same, unchanged original automata. Minterm computation is quick and practically free optimization which can be used every time an intersection of finite automata is computed.

We apply the minterm computation before we start executing any optimized algorithms introduced here or any others. Instead of putting original automata as the input to optimized algorithms, we compute minterms for such automata and substitute all transitions with the generated minterms. The new input to optimized algorithms are automata with minterms which can be easier to work with and their intersection can be computed quickly and more precisely. If we need to use the intersection automaton further, not just to resolve emptiness problem, we simply substitute the minterms in the product with the corresponding original transition symbols back.

Chapter 4

Experiments

The reference implementation[1] of the proposed optimizations, written in Python 3, as well as a complete table of all of our experiments and their results and graphs is publicly accessible on a Codeberg repository[2]. There is further explanation of the following graphs as well as additional graphs with description and in-depth analysis of performed experiments.

Benchmark with sets of different finite automata used on our benchmark problems are available on a GitHub repository[3]. These finite automata are obtained from runs of regular model checking tool on verification of pointer program and parametric protocols created in [3] based on method of abstract regular model checking from [4]. Such verification runs often execute operations similar to emptiness problem or product construction.

Our experiments cover the benchmark automata from 40 various categories of verification runs. In the benchmark, there are in total 5707 finite automata. However, each category contains similar finite automata recognizing similar languages. The results of our experiments on our benchmark problems for different combinations of finite automata from the same category are nearly identical. Thus, we choose representative finite automata from each category randomly.

We test combinations of finite automata from each category to determine the product construction and decide the emptiness of the finite automata intersection. In our experiments, our main objective is to find out how much our optimizations reduce product state space in both our benchmark problems and compare pruning capabilities of our optimizations. We want to know what are the pruning capabilities of both our optimizations. Whtehter they are efficient. Further, we want to compare pruning capabilities of length abstraction and Parikh image abstraction to see whether Parikh image pruning capabilities are higher and by how much.

Our abstractions implemented by the reference implementation are not mature enough to properly measure the time cost of computation yet. Future work includes efficient implementation and further optimizations of our abstractions. Nevertheless, our experiments show our optimizations can sometimes speed up the execution of both benchmark problems.

In this chapter, we present a few experiments which show and compare the pruning capabilities of our optimizations on our benchmark problems. Second, we show what impact have optimizations on our abstractions on computaiton time benchmark problems.

[1]In the reference implementation, we use Z3 as an SMT solver and automata operations are handled by for our purposes modified library Symboliclib.

[2]https://codeberg.org/Adda/optifa

[3]https://github.com/ondrik/automata-benchmarks

4.1 Length Abstraction

In our first experiment, we want to find out what are the pruning capabilities of length abstraction for both our benchmark problems on our benchmark automata. The graph in Figure 4.1 shows a comparison of product state spaces sizes in basic product construction and our optimized algorithm considering length abstraction for emptiness problem. The graph in Figure 4.5 shows shows a comparison of product state spaces sizes ifor basic product and product optimized by length abstraction for product construction.

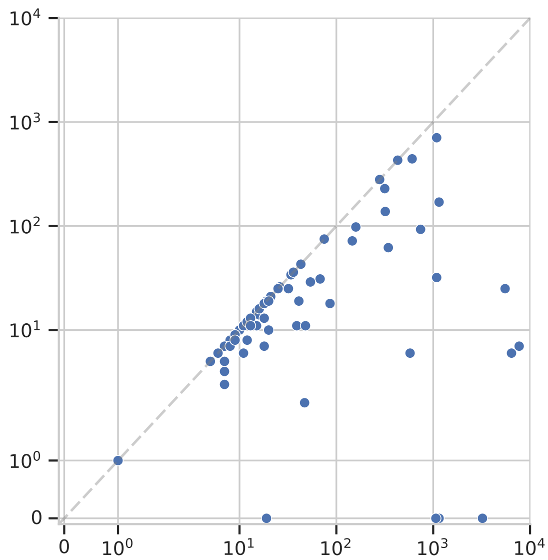

Figure 4.1: Emptiness problem.

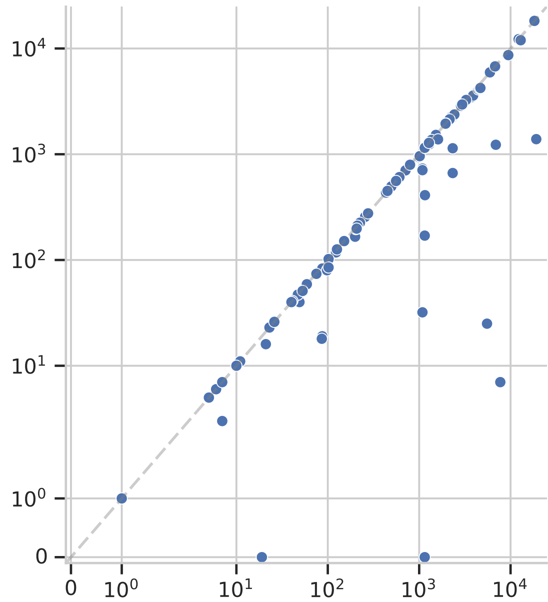

Figure 4.2: Product construction.

Figure 4.3: Comparison of state space sizes by basic and optimized product construction for both benchamark problems. Both axes are in symmetrical logarithmic scale[5], x-axis showing the number of states generated by the naive algorithms, y-axis state space sizes of the optimized algorithms.

As we can see, length abstraction successfully prunes state space in some cases. Notice that for some cases, length abstraction can stop product generation immediately on the first product state.

The results where length abstraction have difficulties with pruning product space are influenced by our benchmark automata. The combinations of benchmark automata in each category rarely have empty intersections. Therefore, for our next experiment, we want to see whether slight modifications of input automata can highlight the strengths of length abstraction. To further extend the set of benchmark automata, to each category, we add finite automata with slight modifications which we combine with original representatives in our experiments. These modifications imitate generation of variations of the same finite automata with different final states (similar to finite automata generated by string solving method from [1]) or little modifications of transitions (removed transitions or changed tran-

[5] Plot is linear around 0 instead of logarithmic.

sition symbols). We want to see whether we can create with our modifications a prunable state space and observe how the length abstraction can notice the difference.

The following graphs show the results of intersection of combinations of the modified benchamrk automaton with the original representative from each category for both deciding the emptiness problem and product construction. The graph in Figure 4.4 shows the comparison of product state spaces sizes in basic and our optimized product construction for emptiness problem. Sorted in order of increasing product state space size of the unoptimized product. The graph in Figure 4.5 shows the comparison of product state spaces sizes in basic and our optimized product construction for product construction.

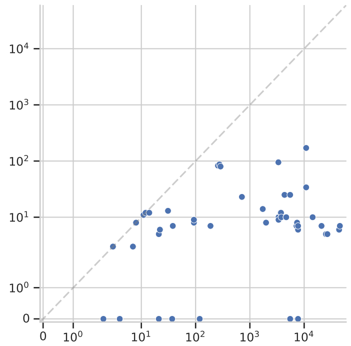

Figure 4.4: Emptiness problem.

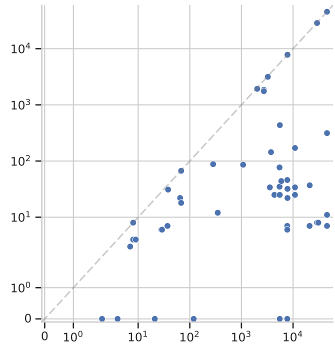

Figure 4.5: Product construction.

Figure 4.6: Comparison of state space sizes generated by basic product and product optimized by length abstraction of both our benchmark problems. Both axes are in symmetrical logarithmic scale, x-axis showing the state space size of basic product, y-axis state space size of the optimized product.

We conclude that length abstraction can usually notice the difference between modified and original automata. Length abstraction is able to prune state space more often and the pruning capabilities for these automata are sufficient. We can see from the graphs that the larger the unoptimized product gets, the higher impact length abstraction has on the product state space size.

It is worth mentioning that we have neglected the number of generated states for our lasso automata. As we can see in Figure 4.9, even when counting with lasso automata states, the total number of generated states in the whole process of the product construction is usually lower than the basic product state space size. The larger the automata are, the better results we get. It is understandable that for smaller original automata the overhead of generating lasso automata is significant in comparison with the small generated product state space sizes. However, the larger the original automata get, the lesser the overhead of the number of lasso automata states is in comparison with the basic product state space.

4.1.1 Length Abstraction Optimization without SMT solver

We optimize evaluation of compatibility of length abstractions by substituting SMT solver with solving linear congruence equations. To show how linear congruences speed up the computation, we present the following experiment.

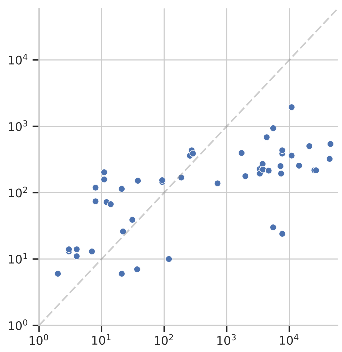

Figure 4.7: Emptiness problem.

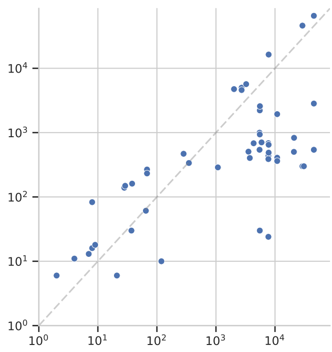

Figure 4.8: Product construction.

Figure 4.9: Comparison of state space sizes in basic product and product optimized by length abstraction with a sum of states generated for both lasso automata. Both axes are in symetrical logarithmic scale, x-axis showing the number of states in the unoptimized product, y-axis the number of states in the optimized product.

The Figure 4.10 shows computation time from our benchmark problems pruned by length abstraction with and without SMT solver.

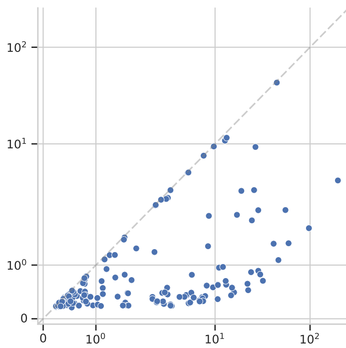

Figure 4.10: Comparison of time consumption of length abstraction evaluated by SMT solver and length abstraction evaluated without SMT solver combining both benchmark problems. Both axis are in symmetrical logaritmic scale. They show time consumption in seconds: x-axis length abstraction evaluated by SMT solver, y-axis length abstraction evaluated without SMT solver.

We can see that computation of both benchmark problems is faster for length abstractions solved by linear congruences. This optimization improves significantly computation time. Thus, whenever we use length abstraction, we should only evaluate compatibility with linear congruences instead of SMT solver.

Out of all experiments with length abstraction, one weakness of length abstraction is clear. The more final states the original automata have, the more difficult it is to optimize product construction using length abstraction. Every final state increases the number of accepted different lengths of automaton. Therefore, with automata where nearly every state is a final state, length abstraction cannot easily determine which product states can be pruned.

4.2 Parikh Image Computation

Length abstraction can sometimes prune product state space significantly, sometimes cannot. We introduced finer state language abstraction using Parikh images. Parikh image abstraction should prune state space more often, even in cases where length abstraction fails. In this section, we show experiments with Parikh image abstractions. First, we are interested in pruning capabilities of Parikh image abstraction. Later, we provide experiment with optimizations of Parikh image abstraction.

We want to find out what are the pruning capabilities of Parikh image abstraction in both our benchmark problems our our benchmark automata. In Figure 4.13, we can see how Parikh image prunes state space.

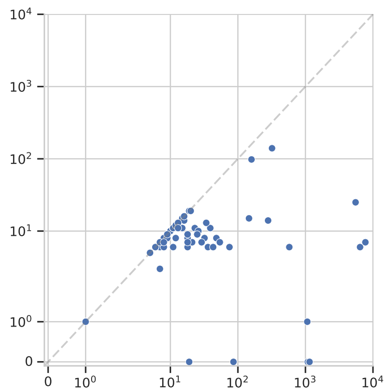

Figure 4.11: Emptiness problem.

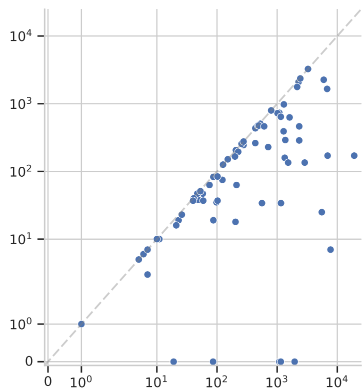

Figure 4.12: Product construction.

Figure 4.13: Comparison of state space sizes generated by basic and optimized product construction with Parikh image abstraction. Both axes are in symmetrical logarithmic scale, showing state space sizes: x-axis of basic product, y-axis of optimized product.

We can see that Parikh image prunes state space significantly in many cases. Notice that for some cases, Parikh image is able to stop product construction immediately for the initial product state.

In the next experiment, we want to find out whether the Parikh image abstraction has higher pruning capabilities than length abstraction. The Figure **??** compares pruning capabilities of length and Parikh image abstractions.

We conclude from the experiment that Parikh image often optimizes the product state space more than length abstraction (at worst products are equal). In many cases, Parikh image optimization is able to prune vast state space by determining incompatible abstractions even if length abstractions are compatible.

Furthermore, notice the dots at the bottom of the graph. Here, Parikh image is able to determine that product language is empty on the first product state and immediately stop the product construction.

Clearly, pruning capabilities of Parikh image abstraction are higher than of length abstraction. Parikh image can prune state space in cases where length abstraction fails to.

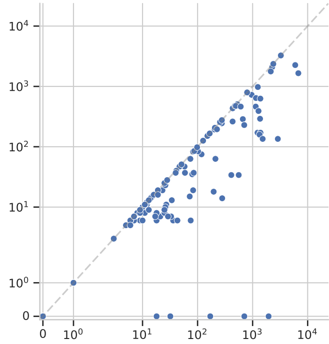

Figure 4.14: Comparison of state spaces generated by length abstraction and Parikh image abstraction combining both benchmark problems. Both axis are in symmetrical logaritmic scale. Axis show product state space sizes: x-axis for length abstraction, y-axis for Parikh image abstraction.

4.2.1 Incremental SMT solving

Incremental SMT solving proves to be a great improvement to the Parikh image computation optimization. The number of conjuncts in Parikh images depends on the number of states in finite automata, the number of transitions and the number of initial or final states. See Table 4.1 for an example comparison of the number of all conjuncts in Parikh image, conjuncts common to all product states (persistent clauses) and state specific conjuncts (state specific clauses).

Product States	All Conjuncts	Persistent Conjuncts	State Specific Conjuncts	Ratio
434	2652	1782	870	67.2%

Table 4.1: An example proportion of persistent and state specific conjuncts in Parikh image computation with incremental SMT solving optimization. *Product States* column shows the number of product states in the whole intersection product, *All Conjuncts* column shows the number of conjuncts in each computed Parikh image, *Persistent Conjuncts* column shows the number of persistent conjuncts in the whole Parikh image (out of the all Parikh image conjuncts), *State Specific Conjuncts* column states how many Parikh image conjuncts have to be recomputed for each product state and *Ratio* column shows the ratio of persistent conjuncts in all Parikh image conjuncts.

For a product of 434 states, each product state Parikh image contains 2652 conjuncts. From those, 1782 conjuncts are persistent conjuncts and the remaining 870 are state specific conjuncts. A proportional ratio o persistent conjuncts in whole Parikh image is around 67.2%. The number of persistent conjuncts (experimentally determined to be usually around 70%) for our benchmark automata means around 70% of each computed Parikh image conjuncts can be precomputed once and used for the whole product generation. Only 30% of conjuncts must be computed repeatedly for each potential product state.

4.2.2 Precise Timeout Selection

For our benchmark automata, we have experimentally concluded the ideal timeout for SMT solver to solve Parikh image abstractions compatibility is around 600ms. This gives SMT solver enough time to compute most incompatible cases while it does not wait too long for the confirmation of satisfiability of Parikh image formulae for compatible cases. Our benchmark automata have however large numbers of transitions from each state, and therefore our timeout might not work best for other types of automata and their complexity. We suggest trying running our optimizations first without any timeout and then, according to the results, adjust the timeout according to the needs of given operations and the complexity of used automata.

4.3 Combination of State Language Abstractions

When we combine length and Parikh image abstraction optimizations in one algorithm, we want to reduce the number of product states Parikh image must be computed for. In Figure 4.15, we can see how many product state are evaluated first by length abstraction, eventually later (if length abstraction resolves length abstraction formule as satisfiable) by Parikh image abstraction to know how many product states can be quickly resolved by length abstraction alone. Picking some examples from the previous results for Parikh image, we provide comparison of pruning capabilities of both abstractions on same automata.

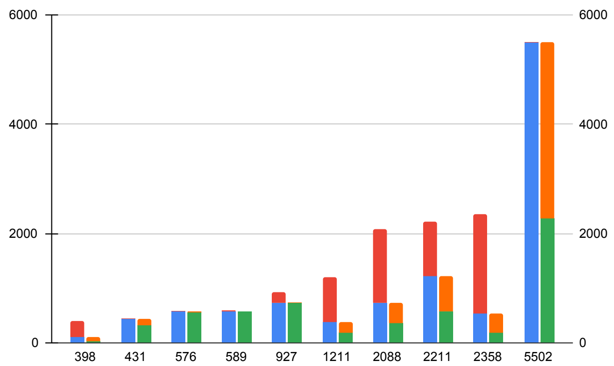

Figure 4.15: Comparison of pruning capabilities of length and Parikh image state language abstractions in optimization algorithm combining both abstractions: x-axis shows the number of processed product states in product construction, y-axis the number of satisfiable and unsatisfiable formulae for these product states. Left stacked column for each example product construction represents length abstraction, right column Parikh image abstraction. Red block represents product states pruned by length abstraction alone, without the need to compute Parikh images and evaluate their compatibility. Blue block represents product states with compatible length abstractions which have to be further examined with Parikh image abstractions. Orange block shows such product states from the blue block which have incompatible Parikh images are pruned by Parikh image, even though length abstraction cannot prune them. Only the product state in the green block are added to the generated product as both length abstraction and Parikh image computation resolved their abstractions as compatible.

4.4 Experiment Results

The experiments shows that our abstractions often prune large parts of the product state space. Length abstraction pruning capabilities are decent, but sometimes it fails to prune state space for intersection of automata with multiple accepted lengths. Pruning capabilities of Parikh image are much higher. Parikh image often succeeds in pruning states where length abstraction fails. Further optimalizations of the abstractions have impact on performance of our abstractions. State language abstractions are combinable without affecting pruning capabilities.

Chapter 5

Conclusion

The most demanding parts of the intersection computation is the generation of product states and transitions of the product automaton. We tried to reduce the size of the generated state space by omitting the states which cannot lead to any final state by deciding the emptiness of such state languages using various state language abstractions over the finite automata states, such as length abstraction using lasso automata or Parikh image computation based on Parikh's theorem. Each approach has been experimentally tested and further optimizations to the proposed algorithms were introduced.

According to our experiments, product state space can be reduced substantially. Pruning capabilities of our abstractions are satisfactory and their optimizations have high impact on computation time. We get great results especially for intersections with long lines or for intersections of automata which differ in accepted lengths. Experiments show our algorithm generates smaller state spaces for both resolution of emptiness problem and product construction. Our abstractions are based on over-approximating abstraction of state languages.

We have concluded that length abstraction is fast and coarse abstraction, Parikh image precise but expensive. Out abstractions can be combined, parrallelized and further extended.

Due to our discoveries, as a future work, we aim to continue with the proposed state language abstractions, optimize their performance with efficient implementation and explore possibilities of additional improvements of these abstractions. We also want to parallelize evaluation of the compatibility of the abstractions.

We have not encountered similar approaches to product construction optimization using length abstraction or Parikh image computation to compare our results with. It might be worth investing into combining our orthogonal approach with other existing algorithms to see how the generated product state space is affected. We are talking about abstraction techniques such as CEGAR [6] and predicate abstraction [7, 16], IMPACT [25], possibly IC3/PDR [18, 5]. All the above techniques have proven efficient in hardware or software verification, and they can be applied in automata too. First attempts to use these techniques in finite automata problem-solving are based on IC3 [19, 28, 9] and on the interpolation-based approach of McMillan [2, 15].

Bibliography

[1] 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.

[2] Amla, N. and McMillan, K. L. Combining Abstraction Refinement and SAT-Based Model Checking. In: *TACAS*. Springer, 2007, vol. 4424, p. 405–419. Lecture Notes in Computer Science.

[3] Bouajjani, A., Habermehl, P., Holík, L., Touili, T. and Vojnar, T. Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O. H. and Ravikumar, B., ed. *Implementation and Applications of Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 57–67. ISBN 978-3-540-70844-5.

[4] Bouajjani, A., Rogalewicz, A., Habermehl, P. and Vojnar, T. Abstract regular tree model checking. In: *ENTCS*. 2006, p. 149(1):37–48.

[5] Bradley, A. R. and Manna, Z. Checking Safety by Inductive Generalization of Counterexamples to Induction. In: *FMCAD*. IEEE Computer Society, 2007, p. 173–180.

[6] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. Counterexample-Guided Abstraction Refinement. In: *CAV*. Springer, 2000, vol. 1855, p. 154–169. Lecture Notes in Computer Science.

[7] Colón, M. and Uribe, T. E. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In: *CAV*. Springer, 1998, vol. 1427, p. 293–304. Lecture Notes in Computer Science.

[8] Conrad, K. *Divisibility and Greatest Common Divisors* [online]. [cit. 2021-08-12]. Available at: https://kconrad.math.uconn.edu/blurbs/ugradnumthy/divgcd.pdf.

[9] Cox, A. and Leasure, J. Model Checking Regular Language Constraints. *CoRR*. 2017, abs/1708.09073.

[10] D'antoni, L. and Veanes, M. Minimization of symbolic automata. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2014, p. 541–553.

[11] Esparza, J. *Automata Theory: An Algorithmic Approach* [online]. 2017 [cit. 2020-10-31]. Available at: https://www7.in.tum.de/~esparza/automatanotes.html.

[12] Esparza, J., Ganty, P., Kiefer, S. and Luttenberger, M. *Parikh's theorem: A simple and direct automaton construction.* June 2011. DOI: 10.1016/j.ipl.2011.03.019.

[13] Fiedor, T., Holík, L., Janku, P., Lengál, O. and Vojnar, T. Lazy Automata Techniques for WS1S. In: *TACAS (1).* 2017, vol. 10205, p. 407–425. Lecture Notes in Computer Science.

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

[15] Gange, G., Navas, J. A., Stuckey, P. J., Søndergaard, H. and Schachte, P. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In: *TACAS.* Springer, 2013, vol. 7795, p. 277–291. Lecture Notes in Computer Science.

[16] Graf, S. and Saïdi, H. Construction of Abstract State Graphs with PVS. In: *CAV.* Springer, 1997, vol. 1254, p. 72–83. Lecture Notes in Computer Science.

[17] Greicius, A. *Linear Congruences* [online]. 2012 [cit. 2021-08-12]. Available at: http://gauss.math.luc.edu/greicius/Math201/Fall2012/Lectures/linear-congruences.article.pdf.

[18] Hoder, K. and Bjørner, N. Generalized Property Directed Reachability. In: *SAT.* Springer, 2012, vol. 7317, p. 157–171. Lecture Notes in Computer Science.

[19] Holík, L., Janku, P., Lin, A. W., Rümmer, P. and Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.* 2018, vol. 2, POPL, p. 4:1–4:32.

[20] Holík, L., Lengál, O., Síč, J., Veanes, M. and Vojnar, T. Simulation Algorithms for Symbolic Automata. In: *Proc. of 16th International Symposium on Automated Technology for Verification and Analysis.* Springer Verlag, 2018, vol. 11138, no. 1, p. 109–125. DOI: 10.1007/978-3-030-01090-4_7. ISBN 978-3-030-01089-8. Available at: https://www.fit.vut.cz/research/publication/11801.

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

[22] Janků, P. and Turoňová, L. Solving String Constraints with Approximate Parikh Image. In: Moreno Díaz, R., Pichler, F. and Quesada Arencibia, A., ed. *Computer Aided Systems Theory – EUROCAST 2019.* Cham: Springer International Publishing, 2020, p. 491–498. ISBN 978-3-030-45093-9.

[23] Kozen, D. C. Parikh's Theorem. In: *Automata and Computability.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1977, p. 201–205. ISBN 978-3-642-85706-5. Available at: https://doi.org/10.1007/978-3-642-85706-5_35.

[24] Lin, A. W. and Barceló, P. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: *POPL.* ACM, 2016, p. 123–136.

[25] McMillan, K. L. Lazy Abstraction with Interpolants. In: *CAV*. Springer, 2006, vol. 4144, p. 123–136. Lecture Notes in Computer Science.

[26] Siegel, S. F. and Yan, Y. Action-Based Model Checking: Logic, Automata, and Reduction. In: *CAV (2)*. Springer, 2020, vol. 12225, p. 77–100. Lecture Notes in Computer Science.

[27] Sipser, M. *Introduction to the Theory of Computation*. 3rdth ed. Cengage Learning, 2013. ISBN 13: 978-1-133-18779-0.

[28] Wang, H., Tsai, T., Lin, C., Yu, F. and Jiang, J. R. String Analysis via Automata Manipulation with Logic Circuit Representation. In: *CAV (1)*. Springer, 2016, vol. 9779, p. 241–260. Lecture Notes in Computer Science.

Appendix A

Contents of the Included Storage Media

The following list shows the contents of the included storage media. Listed are only the folders on the highest levels in the folder hierarchy.

- `optifa/`: The main folder with optimizations of state language abstractions and all related files.
  - `docs/`: The LaTeX source files for this paper.
  - `results/`: The results gained by our experiments.
  - `src/`: The implementation of our optimizations and scripts to run them.
  - `basicDFAs/`: Example finite automata in Timbuk format used in this paper.
- `Symboliclib`: Implementation of the extern library Symboliclib with our modifications included.

Appendix B

Reference Implementation Manual

In order for reference implementation to work, one requires the following programs: Bash[1] or any Bash-compatible shell, Python 3.10[2] or higher, Python library Symboliclib with our modifications and additions[3] and Z3 solver API for Python: Z3Py from Z3 solver repository[4]. Further, to run comparison tests of our optimizations, a command-line benchmarking tool hyperfine[5]. The accepted finite automata file format is Timbuk[6].

Each program can be run with `--help` to show quick help message explaining how to run said program.

You can run tests for all our state language abstractions with `run_tests.sh` as follows:

```
./run_tests.sh -a <finite_automaton_A_file> -b <finite_automaton_B_file> -o <output_file>
```

Separate optimizations can be run with their respective scripts:

- length abstraction with `resolve_satisfiability_length_abstraction.py`, and
- Parikh image abstraction with `resolve_satisfiability_parikh_image.py`.

Combined optimization algorithm using both length and Parikh image abstractions can be run with `resolve_satisfiability_combined.py`.

Each program offers various flags and required or optional arguemements to run adjust the run according to our requirements: Whether to construct full product or just test emptiness of the intersection, which abstraction-specific optimizations to enable, where to store results and the generated product, etc.

Automata with replaced transition by minterms cen be generated with `get_minterms.py`.

[1] https://www.gnu.org/software/bash/
[2] https://www.python.org/
[3] https://codeberg.org/Adda/symboliclib/; Remember to add Symboliclib to Python path.
[4] https://github.com/Z3Prover/z3; Remember to add Z3Py API to Python path.
[5] https://github.com/sharkdp/hyperfine
[6] https://gitlab.inria.fr/regular-pv/timbuk/timbuk/-/wikis/Specification-File-Format

Appendix C

Complete Optimization Algorithm

Input : NFA $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$, NFA $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$
Output: NFA $P = (A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

$Q, \delta, F \leftarrow \varnothing$
$I \leftarrow I_1 \times I_2$
$W \leftarrow I$
$res \leftarrow False$
$solved \leftarrow \varnothing$
`addPersistentClauses()`
while $W \neq \varnothing$ **do**
    picklast $[q_1, q_2]$ **from** W
    add $[q_1, q_2]$ **to** $solved$
    if `skippable(`$[q_1, q_2]$`)` **then**
        $res \leftarrow True$
    else
        if $\alpha^{LA}(q_1) \wedge \alpha^{LA}(q_2)$ ***is*** unsat **then**
            $res \leftarrow False$
        else
            `smtSolverPush()`
            `addStateSpecificClauses(`$[q_1, q_2]$`)`
            $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat
            `smtSolverPop()`
            if $res = Unknown$ **then**
                $res \leftarrow True$
    if $res = True$ **then**
        add $[q_1, q_2]$ **to** Q
        if $q_1 \in F_1$ ***and*** $q_2 \in F_2$ **then**
            add $[q_1, q_2]$ **to** F
        forall $a \in \Sigma$ **do**
            forall $q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)$ **do**
                if $[q_1', q_2'] \notin solved$ ***and*** $[q_1', q_2'] \notin W$ **then**
                    add $[q_1', q_2']$ **to** W
                add $[q_1', q_2']$ **to** $\delta([q_1, q_2], a)$

Algorithm 11: Product construction algorithm using both length abstraction and Parikh image computation and all their optimizations.