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

Abstract

We explore possibilities of using various abstractions of automata languages in optimization of automata algorithms used in mathematics, computation theory and logic. We focus on abstracting languages of states to sets of possible word lengths or Parikh images, represented as semi-linear sets, and exploring options of using them to optimize the construction of result of automata operations by pruning pairs of states with incompatible abstractions. We continue towards optimization of these techniques.

We use synchronous product construction and its emptiness test as our benchmarking operation on automata in our experiments. Nevertheless, our abstractions are applicable on many other typical automata operations, e.g., complement generation etc.

Abstrakt

Objevujeme možnosti použití různých abstrakcí jazyků automatů pro optimalizaci automatových algoritmů používaných v matematice, výpočetní teorii a logice. Zajímáme se o abstrakci jazyků stavů na množiny možných délek slov nebo Parikovy obrazy, reprezentované jako semi-lineární množiny, a zkoumáme možnosti jejich využití k optimalizaci konstrukce výsledku automatových operací pomocí odstraňování stavů s nekompatibilními abstrakcemi. Následuje optimalizace těchto technik.

Používáme synchronní konstrukci průniku a test jeho prázdnosti jako operaci pro experimentální vyhodnocení metod. Nicméně naše abstrakce jsou aplikovatelné na mnohé typické automatové operace, například generaci doplňku aj.

Keywords

Finite Automata, State Languages Abstraction, SMT solving, Product Construction, Emptiness Test, Intersection Computation Optimization, State Space Reduction, SMT Solving, Length Abstraction, Parikh Image, Mintermization

Klíčová slova

Konečné automaty, Abstrakce jazyků stavů, SMT výpočty, Konstrukce produktu, Test prázdnosti, Optimalizace výpočtu průniku, Redukce stavového prostoru, Délková abstrakce, Parikovské 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

awdawdDo tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

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 2, 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. Automata are commonly used in mathematics and computation theory in general (e.g., in model checking [18] or string solving and analysis [16]). Their usage in the field of logic is just as important, too (e.g., WS1S [9, 10]). Finite automata are conceptually straightforward. However, operations on finite automata can produce extensively larger and harder to work with result automata. Such operations 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 explore possibilities of using various abstractions of automata languages in optimization of automata algorithms. We will study different approaches to abstraction of languages of states. We 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 result automata by pruning pairs of states with incompatible abstractions. We continue towards optimization of these techniques. Furthermore, we consider the use of mintermization and other approaches to further improve these methods.

We consider several usual operations which take lots of computational time and generate vast state space as a result. One of such operations is the construction of finite automata intersection generated by the synchronous product construction algorithm. We use product construction and its emptiness test as a benchmarking operation in our experiments for evaluation of our methods. Nevertheless, our optimization methods are generally usable on many other typical automata algorithms. Consequently, even if our approaches to the optimization problems are introduced on product construction algorithm, our discoveries have wider impact and are in some form applicable on other automata operations, e.g., complement generation etc.

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 expensive on computational time. Furthermore, the generated product state space increases exponentially according to the number of used automata and the number of their states[1]. However, there are often large parts of the generated state space which cannot accept any words (no final states can

[1]The product construction sometimes *explodes* in a huge product state space.

be reached from these states), yet are still generated. Therefore, it is important to have a decent algorithm to minimize the generated product state space as much as possible.

In our methods, we focus mainly on decision-making about the satisfiability problem—solving the emptiness of the intersection of finite automata. We try to identify which generated product states cannot lead to any accept states and do not continue from 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 successor states, pruning the generated state space.

We start with length abstraction of state languages[2] optimization. We have computed possible lengths of accepted words for each automaton and their states. Length abstraction is an effective and simple method but the pruning capabilities are limited to specific qualities. Length abstraction alone cannot sometimes detect unnecessary state space for automata with rich alphabet and many transitions from each state for such state languages accept multitude of words lengths.

When the lengths of words recognized by the languages of the current states are not compatible with each other—the original languages of the corresponding states cannot accept a word of the same lengths—there is definitely no transition from this product state leading to accepting the same word in both original automata. We can prune such states from our generated product. Consequently, this removes the need to even consider their potential successor states, which are generated normally. We trim the generated product to only states whose corresponding original states languages can accept words of the same lengths. Even though there might still be states which do not lead to any accepting state in the final product, this simple optimization already trims a substantial parts of the normally generated synchronous product state space reasonably often.

The second optimization approach we consider is the computation of Parikh images[3] for states in a potential product state. Parikh image of a word tells us how many times a symbol occurs in a word[4]. Parikh image of a language is then a semi-linear formula describing the relation between the number of symbol occurrences in words in a language. In contrast to the length abstraction, we have additional information about the product states (the number of symbols in words). We can more precisely identify unnecessary state space by determining compatibility of Parikh image abstractions. However, the Parikh image computation is an expensive operation.

It is necessary to decide whether the trade-off of unoptimized basic algorithm generating larger product state space requiring less computation time for reduced product state space generated by our optimized algorithm using Parikh images with additional computation time requirements is worth our attention. For certain operations over the automata, the product state space size is crucial. Considering we may need to work with the same product multiple times or simply need to execute a single operation on the product[5], reduced state space can spare extensive amounts of computation time further down the processing line. Furthermore, generating smaller state space using our Parikh image optimization can improve computation time for the sole product generation algorithm in case substantial parts of otherwise generated state space are pruned or even when the whole product is proved

[2] to sets of possible word lengths

[3] represented as semi-linear sets

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

[5] Even more so if automata operations are chained one after another, each operation increasing the complexity of the previous one.

to be empty, which can be quickly determined by our optimization *on the fly*, whereas the classic unoptimized algorithm would proceed to generate the whole suppositional product with useless fragments only to find in the end that the product is empty.

Our another optimization uses mintermization 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 in less computation time.

We have implemented these optimizations and experimented with several different automata, tried various combinations of them, generated their products and tried to solve their emptiness test, focusing mainly on the number of trimmed product states in the process. For certain types of automata of certain qualities, these optimizations works really well. Parikh image abstraction usually trims vast state spaces where length abstraction cannot prune anything and basic product state space explodes (e.g., from 20000 to 10 product states). In addition, it is successful at immediately stopping product generation if the product is empty.

The contribution of this work can be summarized as follows:

1. heuristics trimming the generated state space of finite automata operations based on length abstraction, Parikh image computation, mintermization, and

2. 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 [7] or [19].

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 the* ***initial state****, and*
- $F \subseteq Q$ *is a* ***set of final (accept) states****.*

A *run* of A on input $a_0a_1a_2 \dots a_{n-1}$ 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 $0 \leq i \leq n$, $q_0 = I$ and $\delta(q_i, a_i) = q_{i+1}$ for $0 \leq i \leq n-1$. A run is *accepting* if $q_n \in F$. The automaton A *accepts* a word $w \in \Sigma^*$ if it has an accepting run on input w. A *language* recognized by finite automaton A is a set $L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$. A single *transition* from transition function 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 accept 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****:* $\delta : Q \times \Sigma_\epsilon \to P(Q)$*, where* $\Sigma_\epsilon = \Sigma \cup \epsilon$ *and* $P(Q) = \{R \mid R \subseteq Q\}$ *is a set of subsets of* Q*, and*
- $I = \{q \mid q \in Q\}$ *is a non-empty* ***set of initial states****.*

[1] No accept 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 and B are said to be *equivalent* when both accept the same language: $L(A) = L(B)$.

For every NFA A exists a corresponding equivalent DFA B. *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) *Operations on automata A_1 and A_2 yield a result—a product A as a 5-tuple deterministic finite automaton $A = (Q, \Sigma, \delta, I, F)$.*

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, we can define:

- *a set of states $Q = Q_1 \times Q_2$,*
- *a transition relation $\delta : Q \times \Sigma \to P(Q)$,*
- *a set of initial states $I = I_1 \times I_2$, and*
- *a set of accepting states $F = F_1 \times F_2$.*

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 mainly 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 resulting automaton language: $L(A)$ does not accept any words.

We work with a basic product construction algorithm 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, $\forall p \in P : p \leq \gamma(\alpha(p))$. 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 $(A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

```
1  Q, δ, F ← ∅
2  I ← I_1 × I_2
3  W ← I
4  while W ≠ ∅ do
5      pick [q_1, q_2] from W
6      add [q_1, q_2] to Q
7      if q_1 ∈ F_1 and q_2 ∈ F_2 then
8          add [q_1, q_2] to F
9      forall a ∈ Σ do
10         forall q_1' ∈ δ_1(q_1, a), q_2' ∈ δ_2(q_2, a) do
11             if [q_1', q_2'] ∉ Q then
12                 add [q_1', q_2'] to W
13             add [q_1', q_2'] to δ([q_1, q_2], a)
```

Algorithm 1: Classic product construction algorithm used as a base for our optimization methods which extend the algorithm by deciding compatibility of state language abstractions.

Chapter 3

State Language Abstractions

In this chapter, we intruduce several state language abstractions. We aim to optimize operations on finite automata such as product construction, complement computation, minimalization or determinization and inclusion test. Furthermore, we want to create state language abstractions which can work for different automata structures: operations on transducers, operations with alternating automata such as its emptiness or a conversion of an alternating automaton to its NFA representation, conversion of finite automata to flat automata, etc. Our optimization methods therefore have to be able to improve substantially important processes used throughout multitude of theoretical areas as well as in praxis.

We try to optimize automata operations with our optimizations methods. For the purpose of introducing our methods, we will focus solely on synchronous product construction of automata intersection from now on. However, the proposed optimizations can be applied to other operations as well.

When constructing a product, a considerate number of generated product states are non-terminating and thus unnecessary. Furthermore, the whole product must be constructed before we can determine whether the automata intersection is empty. 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 generation algorithm. 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 will consider only two automata (A_1 and A_2). The concept of state language abstractions is to find a state language abstraction $\alpha^X(q)$ of a state q in abstraction X ($\alpha^{LA}(q)$ for length abstraction and $\alpha^{PI}(q)$ for Parikh image abstraction) representing a formula in first-order predicate logic. Both our $\alpha^{LA}(q)$ and $\alpha^{PI}(q)$ respect Galois connection. Hence, they are an over-approximation of state language of q. We compare such state abstractions in different finite automata ($\alpha(q_1)$ where $q_1 \in Q_{A_1}$, $\alpha(q_2)$ where $q_2 \in Q_{A_2}$) to find out whether they are compatible with each other. If not, we can assume the corresponding state languages are neither and can prune such product state $p = [q_1, q_2]$.

Because our proposed abstractions are always over-abstractions, they cannot accidentally trim any product states leading to an accepting state (cannot change the intersection language). The optimized product language is the same as the one generated from the naive

product construction algorithm. Consequently, is it completely safe to use our optimizations with any kind of automata for any kind of uses.

We perform experiments with the following optimization methods mostly on product construction of two NFAs, improving the naive product construction algorithm to generate optimized products. We chose product construction as our benchmarking operation on automata for its straightforwardness allowing us to clearly follow pruning capabilities of our proposed optimization methods, even though the naive product construction algorithm complexity can be at most *only* quadratic. Nevertheless, our optimization techniques are to be used in various fields of automata theory and to allow us to optimize even other, more complex problems (determinization, minimization, emptiness of alternating automata, ...).

3.1 Length Abstraction of State Languages

Our task is to try to minimize the number of generated states when trying to resolve the product construction of automata intersection and test its emptiness. One possible solution is looking for lengths of words accepted by both automata—testing whether both automata recognize words of the same lengths. Afterwards, we check the original transition symbols for generating new product states[1]. Consequently, we can resolve the emptiness test of some intersections very quickly and optionally optimize the product construction, when we need to generate the whole product.

We will explain our chosen approach to the problem of optimizing product construction and deciding its emptiness test using length abstraction α^{LA}, but first some rudimentary knowledge on length abstraction is required.

3.1.1 Length Abstraction Represented by Lasso Automata

Our chosen approach to the problem of optimizing product construction and deciding its emptiness test includes using length abstraction over the finite automata to 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.

α^{LA} generalizes the language recognized by the NFA A by considering only the possible lengths of words accepted by A. It is an over-approximation of $L(A)$. This means that if a word is not accepted by the length abstraction of A, it cannot be accepted by A, either.

Computing length abstraction over the languages of finite automata (and over individual states in the automata in particular) 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 single *loop* (resolving the cycles in the original automaton) resembling a lasso with a few accepting states representing the lengths of accepted words. Thus, the name is a lasso automaton.

Let us demonstrate the construction of $LSA(A_1)$ for the following NFA A_1, which we continue to use to introduce our length abstraction optimization. You can create $LSA(A_1)$ by taking A_1, considering all transition symbols as a single transition symbol and determinizing the result. $LSA(A_1)$ is an automaton accepting every length of any word recognized by $L(A_1)$. Consequently, it is easy to compute semi-linear set (formulae[2]) for the allowed lengths of words, which can be effectively evaluated. We are computing these formulae for

[1] So we do not get non-empty intersection results when there is no word both original automata actually accept and only their lengths correspond.

[2] disjunction of linear equations

individual product states (precisely for the corresponding states in the original automata), checking their satisfiability and consequently construct only those product states for which the length abstraction compatibility test resolves as satisfiable.

$$A_1 = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{0, 1\}, \delta_1, \{q_0\}, \{q_4\})$$

Transition relation δ_1 is depicted in Figure 3.1.

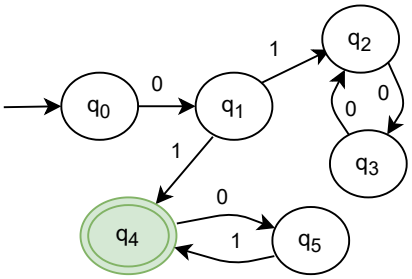

Figure 3.1: Non-deterministic finite automaton A_1.

NFA A_1 is a non-deterministic finite automaton (see state q_1) and uses multiple input symbols. Due to the fact that we work only with recognized word lengths, we can substitute automaton alphabet with unary alphabet of single input symbol $*$[3]. Then, we can compute $LSA(A_1)$ for A_1 with unary alphabet, which is its deterministic equivalent.

$$\Sigma = \{0, 1\} \longrightarrow \Sigma' = \{*\}$$

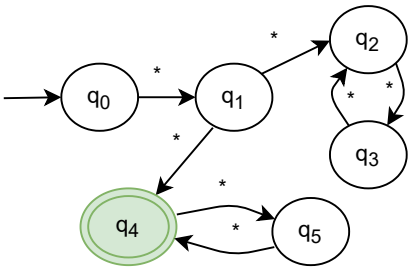

Figure 3.2: Non-deterministic finite automaton A_1 with unified transition symbols.

We start the determinization process on A_1 with unary alphabet. For the final $LSA(A_1)$, see Figure 3.3. $LSA(A_1)$ accepts any words of lengths of words recognized by A_1. We will use these lengths in the process of constructing the product.[4]

[3]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 transition symbols are not significant for our optimization algorithm.

[4]You can notice $LSA(A_1)$ looks different to what is depicted in Section 3.1.1. It is caused by our optimization with generating only a single LSA per original automaton. $LSA(A_1)$ is valid and the fact there are actually two separate automata with one even being inaccessible does not raise any issue for us. The reason for this behaviour will be further explained in Section 3.1.2.

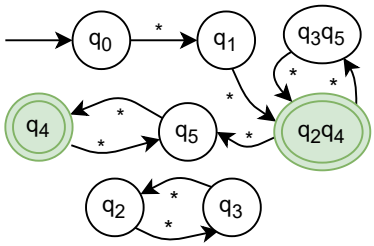

Figure 3.3: Lasso automaton $LSA(A_1)$ for the original NFA A_2.

3.1.2 Single Lasso Automaton for Each Original Automaton

When we are constructing a product P, we do not want to regenerate LSA for each new product state p. This is inefficient. Therefore, our algorithm generates LSA only once—state by state—checking every time, whether the new state q is not already present in Q_{LSA}.

Due to the nature of LSA, the successive product states p' generate LSA very similar to LSA(p). We just need to append new states to Q_{LSA}. As a result, we will work with only two lasso automata (possibly with multiple loops and/or multiple handles)—one for both automata whose intersection is computed.

If $q \notin Q_{LSA}$, we add q to Q_{LSA} and continue with the following states q' until we either create an entirely new loop in LSA or generate $q' \in Q_{LSA}$. If $q' \in Q_{LSA}$, we can stop generating q' as from now on, $\forall q' : q' \in Q_{LSA}$.

3.2 Product Construction with Length Abstraction Optimization

The core of the product construction algorithm remains unchanged, but there are a few differences. The Algorithm 2 shows how we alternate the original product construction algorithm to optimize the algorithm and resolve emptiness test for each *branch* of the potential product automaton.

We will call W from line 3 a work set. It stores the potential product states prepared for testing for satisfiability and other processing, which we pick from W one by one[5].

The optimization process starts when a product state p is picked from W. Instead of immediately generating new successive product states p', we test p for satisfiability of length constraints of recognized words from p: $sat(\Phi^{LA}(p))$ 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 $sat(\Phi^{LA}(p))$. If $sat(\Phi^{LA}(p))$, i.e., there will be an accepting run using p (see line 22), we add p to Q, possibly to F and generate p'.

[5]In spite of the fact that more approaches are valid, we strongly recommend picking the last added product state from the work set W (see line 8)—using Depth-first Search for a graph algorithm—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 with implemented satisfiable state skipping optimization, explained in Section 3.2.1.

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 $(A_1 \cap A_2) = (Q, \Sigma, \delta, I, F)$ with $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

```
1  Q, δ, F ← ∅
2  I ← I_1 × I_2
3  W ← I
4  res ← False
5  solved ← ∅
6  while W ≠ ∅ do
7      picklast [q_1, q_2] from W
8      add [q_1, q_2] to solved
9      res ← α^LA(q_1) ∧ α^LA(q_2) is sat
10     if res = True then
11         add [q_1, q_2] to Q
12         if q_1 ∈ F_1 and q_2 ∈ F_2 then
13             add [q_1, q_2] to F
14         forall a ∈ Σ do
15             forall q'_1 ∈ δ_1(q_1, a), q'_2 ∈ δ_2(q_2, a) do
16                 if [q'_1, q'_2] ∉ solved and [q'_1, q'_2] ∉ W then
17                     add [q'_1, q'_2] to W
18                 add [q'_1, q'_2] to δ([q_1, q_2], a)
```

Algorithm 2: Product construction with length abstraction.

The formulae are generated using $LSA(A_1)$ and $LSA(A_2)$. For every state, we get one or more formulae 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 accepting state, and l is the length of a loop to return to this particular accepting state going through the loop. k is the number of cycles through the loop states until a word ends in an accepting state. When multiple depicted formulae are present (because there are more accepting states in LSA), we append these formulae with *logical or* ($\vee$), then compare these with the formulae from the other LSA for the other initial finite automaton using SMT solver.

To better demonstrate our solution, the second automaton we will be working with is a NFA A_2 from Figure 3.4.

$$A_2 = (\{s_0, s_1, s_2, s_3\}, \{0, 1\}, \delta_2, \{s_0\}, \{s_3\})$$

Transition relation δ_2 is depicted in Figure 3.4.

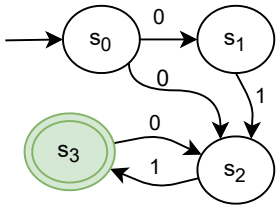

Figure 3.4: Non-deterministic finite automaton A_2.

In Figure 3.5, there is $LSA(A_2)$, which we will be using together with $LSA(A_1)$ shown in Figure 3.3 for computation of recognized word lengths.

```
1  Function isLengthAbstractionSatisfiable(formulaeForFA1, formulaeForFA2):
       Data: Input length formulae of potential product state we are solving the satisfiability for
             (for all possible accept states combinations).
       formulaeForFA1: Formulae for the first finite automaton,
       formulaeForFA2: Formulae for the second finite automaton.
       Result: bool: True if satisfiable, False otherwise.
2      smtAdd(VariableForFormulaForFA1 ≥ 0, VariableForFormulaForFA2 ≥ 0)
3      for formulaForFA1 ∈ formulaeForFA1 do
4          for formulaForFA2 ∈ formulaeForFA2 do
5              smtPush()
6              smtAdd(formulaForFA1.handle + formulaForFA1.lasso ∗
                VariableForFormulaForFA1 =
                formulaForFA2.handle + formulaForFA2.lasso ∗ VariableForFormulaForFA2)
7              sat ← smtCheck()
8              if sat or sat = unknown then
9                  return True
10             smtPop()
11     return False
```

Algorithm 3: Check satisfiability using length abstraction algorithm with SMT solver.

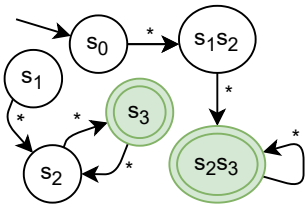

Figure 3.5: Lasso automaton $LSA(A_2)$ for the original NFA A_2.

For A_1 for the initial state (we start computing lengths as if the state q_0 is the initial state) from $LSA(A_1)$, we get existential formula φ[6]. For A_2 for the initial state s_0 from $LSA(A_2)$, we get formula ψ[7].

$$\varphi : \exists k(|w| = 2 \vee |w| = 4 + 2 \cdot k)$$
$$\psi : \exists m(|w| = 2 + 1 \cdot m)$$

When we compare φ and ψ, we get:

$$\exists k \exists m(2 \vee 4 + 2 \cdot k = 2 + 1 \cdot m)$$

We try to find values of k and m such that some of the expressions on the left and on the right side of the equation are equal. We pass this equation to SMT solver to solve its satisfiability. Returns *sat* when satisfiable (*sat* is set to *True*) and *unsat* when unsatisfiable (*sat* is set to *False*). If *unsat* is returned, we can stop generating this *branch* of a NFA as we know for sure there cannot be a word which is accepted by both A_1 and A_2, when there is even no word fulfilling the length requirements. In this case, we have successfully reduced the generated state space by omitting product state $p = [q_0, s_0]$ and any further product

[6] This formula consists of two independent formulae describing there are more possible lengths for accepted words from the same initial state (leading to two independent accepting states in the automaton).

[7] We are using variable m here instead of k to emphasize variables from different formulae are not dependent on each other—they correspond to various accepting states.

states p', which would be later normally generated from p and its successors (assuming the transition symbols correspond).

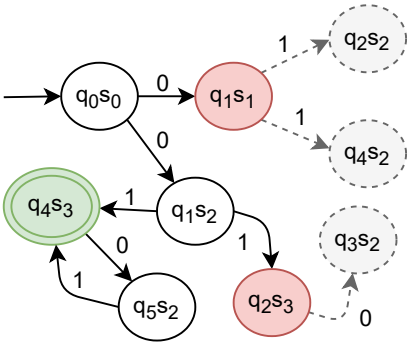

Figure 3.6: Constructed product automaton with depiction of our optimization.

In Figure 3.6, we can see the product of A_1 and A_2 being constructed using our optimization. Red states represent tested states that are resolved as unsatisfiable for computed length formulae and therefore the algorithm omits any successive product states—dashed states (such as q_4s_2 or q_3s_2), which are generated in the basic naive product construction algorithm. The green state is satisfiable and also represents accepting states in both automata. Here, we have found a possible solution accepted by both original automata. If we desire to resolve only the product emptiness test, we can stop the execution of the algorithm here as we have found one accepting state—automata have non-empty intersection.

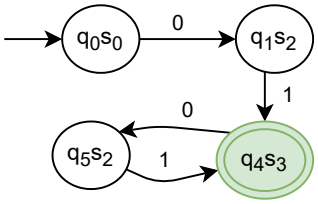

Figure 3.7: Final product generated by our product construction algorithm optimizing its process by omitting unnecessary states on the fly.

As you can notice in Figure 3.7, the product generated by our algorithm has only 4 product states in comparison to 9 product states generated by the classic algorithm.

When we get $\neg sat(\Phi^{LA}(p))$, we do not need to construct any following states p' and check whether $sat(\Phi^{LA}(p'))$ and therefore, we are able to determine whether such *branches* of automata have an empty intersection and we do not need to consider them in the product construction. The emptiness test is successfully accomplished, and we determined that for this *branch* there cannot exist any word accepted by both automata and consequently by their intersection. When we get unsatisfiable result for every branch of the automaton (i.e., no *branch* can lead to any accepting state) even if by inspecting transition symbols it looks like there could be a non-empty intersection, we can say that such input automata have an empty intersection and product construction will be very quick in that case—this is where our optimization dominates.

A note of caution. It is important to understand that we are working only with possible word lengths and therefore when we test the emptiness of intersection of automata, we can resolve 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 $\neg sat(\Phi^{LA}(p))$.

3.2.1 Optimization with Skipping Satisfiable States

When we take new p from W and check whether $sat(\Phi^{LA}(p))$, it is time to add to W all of the possible successive product states p'. When p generates only a single p', we can say with certainty that $sat(\Phi^{LA}(p'))$ too as there is only a single branch in the automaton leading from p to an accepting 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 $LSA(A_1)$ and $LSA(A_2)$.

If A_1 or A_2 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. In Algorithm 4 is depicted the application of skipping satisfiable states. The line 9 from Algorithm 2 is substituted with the contents of Algorithm 4.

```
1 if not skippable([q1, q2]) then
2 |   res ← α^LA(q1) ∧ α^LA(q2) is sat
3 else
4 |   res ← True
```

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

The only change is a test for every checked product state q, which decides whether q can be skipped, if it cannot give us any information which we do not have yet. You can see that we proceed with SMT solver satisfiability check only for q which are generated from the product states with multiple transitions generating q and at least one more product state (in general at least two new potential product states). If only one q is generated, we skip the satisfiability check for q 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 3.8 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, 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 accepting state resolving in *sat* check for q_4s_3 actually.

When we have a series of such states, though, 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 of the initial states of each new branch and then either

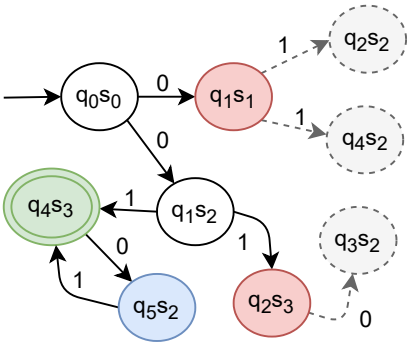

Figure 3.8: Constructed product automaton with depiction of skipping states optimization

omit the entire branch (if *unsat* is returned) or skip checking satisfiability in the entire branch (if *sat* is returned).

3.2.2 Resolving Length Abstraction Satisfiability without SMT Solver

```
1  Function isLengthAbstractionSatisfiable(formulaeForFA1, formulaeForFA2):
       Data: Input length formulae of potential product state we are solving the satisfiability for
             (for all possible accept states combinations).
       formulaeForFA1: Formulae for the first finite automaton,
       formulaeForFA2: Formulae for the second finite automaton.
       Result: bool: True if satisfiable, False otherwise.
2      for formulaForFA1 ∈ formulaeForFA1 do
3          for formulaForFA2 ∈ formulaeForFA2 do
4              if formulaForFA1.handle = formulaForFA2.handle then
5                  return True
6              else if formulaForFA1.handle > formulaForFA2.handle then
7                  sat ← solveForOneHandleLonger(formulaForFA1, formulaForFA2)
8                  if sat then
9                      return True
10             else
11                 sat ← solveForOneHandleLonger(formulaForFA2, formulaForFA1)
12                 if sat then
13                     return True
14     return False
```

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

3.3 Parikh Image State Language Abstraction

The $\alpha^{LA}(q)$ is a simple and fast optimization abstracting accepted words to only their lengths, but can be too general to detect non-terminating states in some cases. In this section, we present a product construction optimization using Parikh image state language abstraction which aims to make the abstraction more accurate to prune larger quantities of unnecessary generated product state space.

Parikh images provide more information about the finite automata than simple length abstraction as Parikh image abstracts accepted words to numbers of occurrences of transition symbols in words regardless of their position in words instead of only word lengths

```
1  Function solveForOneHandleLonger(formulaForFA1, formulaForFA2):
       Data: Input length formulae of potential product state we are solving the satisfiability for
             (for concrete accept states combination).
       formulaForFA1: Formula for the first finite automaton,
       formulaForFA2: Formula for the second finite automaton.
       Result: bool: True if satisfiable, False otherwise.
2      formulaForFA1.handle ← formulaForFA1.handle − formulaForFA2.handle
3      formulaForFA2.handle ← 0
4      if formulaForFA1.lasso = 0 and formulaForFA2.lasso = 0 then
5          return False
6      else if formulaForFA2.handle = 0 then
7          return False
8      else if formulaForFA1.lasso = 0 then
9          currentIteration ← 0
10         while currentIteration ≤ formulaForFA1.handle do
11             if currentIteration = formulaForFA1.handle then
12                 return True
13             else
14                 currentIteration ← currentIteration + formulaForFA2.lasso
15         return False
16     else
17         gcd ← getGCD(formulaForFA1.lasso, formulaForFA2.lasso)
18         if gcd = 1 then
19             return True
20         else
21             y ← −formulaForFA1.handle
22             while y < gcd do
23                 y ← y + formulaForFA1.lasso
24             if y mod gcd = 0 then
25                 return True
26             else
27                 return False
28     return False
```

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

without consideration of which transition symbols are actually used. Parikh image abstraction allows us to more precisely determine the emptiness of finite automata intersection. However, the Parikh image computation itself consumes a considerate amount of computational time for some of the more extensive finite automata. The question is, whether the added computation time compensates for more precise product generation with higher product states pruning capabilities.

We introduce an algorithm for Parikh image computation applied on each potential product state $p = [q_1, q_2]$ to decide the compatibility of its $\alpha^{PI}(q_1)$, $\alpha^{PI}(q_2)$ (mutual satisfiability of formulae describing the abstractions). If the abstractions are proved to be compatible, p is added to the generated product. Otherwise, p is omitted and no additional p' such that $p' = \delta(p, a)$ accessible only from p are added to the queue to test their abstractions compatibility. Generalization to n-tuples is then a matter of adding additional abstraction equal to the number of input automata.

3.3.1 Parikh Image

We derive our Parikh image construction from the Parikh's theorem [15] described in [8], 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 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.

Our Parikh image formulae consist of the following constraints, in conjunctive normal form. For each potential product state, there exists exactly one our formula of Parikh image describing its regular language. We ask the SMT solver whether the Parikh image constraints for corresponding states in the original automata (one state per automaton) are compatible with each other. This ensures that we construct only those product states which satisfy the Parikh image constraints, otherwise we deem such potential product states redundant and such states can be pruned.

Given an NFA $A = (Q, \Sigma, \Delta, I, F)$ where I is a singleton $I = \{q_0\}$, Parikh image formula φ (as described in [**?**] for solving string constraints) consists of the following conjuncts. φ describes runs of A (precisely, their over-approximation). The defined variables represent qualities of each run, their precise values the precise qualities of the specific run. Satisfiable assignment defines a set of runs with qualities given by the assigned variable values.

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 again 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 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). The constraint $\#_a = \sum_{t=(q,a,q') \in \Delta} y_t$ ensures $\#_a$ is consistent with the number of used t with a.

5. Last, but not least, we make sure the regular language expressed by Parikh image preserves the connectedness of A—the used automata states are accessible from I and they are connected by transitions. Variable z_q for each $q \in Q$ is introduced. z_q represents the length of the path from I to q in a spanning tree of the subgraph with $y_t \geq 0$.

   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.

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.

For SMT solving, it is paramount that we have formulae without universal quantifiers, otherwise the SMT solver computation could *explode* computation time-wise. SMT solver work best with quantifier-free or existential formulae. Thanks to how Parikh image is constructed, our approach takes advantage of these SMT qualities and our Parikh image formulae can be inserted in SMT solver as quantifier-free.

Reduced Parikh Image

The presented Parikh image works as intended. Nevertheless, the described Parikh image computation requires extensive resources and computation time. However, we use Parikh image only for determining the emptiness of the product. Given that most of the computation time is taken by the evaluation of these conjuncts, we try to minimize the number of Parikh image formula conjuncts SMT solver needs to evaluate for each φ.

Consequently, we infer our reduced Parikh image from the above shown Parikh image to further optimize Parikh image computation. We strip Parikh image of for our purposes unnecessary constraints and unify initial states as well as accept states.

Our reduced Parikh image consists of the following conjuncts:

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 4.

3. However, we completely omit the conjuncts for z_q which ensure the connectedness of the Parikh image representation of finite automaton. The reason is that, as we have found out, the difference in pruning capabilities of Parikh image with or without the conjuncts 5 on our benchmark automata is insignificant in comparison to the computation time spared by removing these conjuncts

   The reason conjuncts 5 are so demanding computation time-wise is that all these conjuncts have to be always recomputed for each single state Parikh image is computed for. Furthermore, the conjuncts themselves are complex for even simple automata. For that reason, SMT solvers need extensive resources to compute Parikh images with these conjuncts in consideration.

   Even then, if we require ensuring that the reduced Parikh image represents the connectedness of the finite automaton, we can include these conjuncts, but, thanks to our unification of initial and accept states, we change them as follows to reflect our initial and accept state unification changes:

   The constraint for when q is an initial state ($z_q = 1 \wedge y_t \geq 0$) remains unchanged. However, for every other state, we remove the possibility of $y_t = 0$ and $z_{q'} = 0$ in the second half of the conjuncts. The conjuncts look like this:

$$(z_q = 0 \wedge \bigwedge_{t \in \Delta_q^+} y_t = 0) \vee \bigvee_{t \in \Delta_q^+} (y_t > 0 \wedge z_{q'} > 0 \wedge z_q = z_{q'} + 1).$$

Our goal is to reduce the number of conjuncts the SMT solver needs to compute for each potential product-state. We focus on several optimizations such as incremental SMT solving,

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

Compatibility of Multiple Parikh Image State Language Abstractions

So far, we have shown how to compute Parikh image for a single finite automaton to represent said automaton with a single formula. We want to use this formula in such a way that would allow us to decide satisfiability of those formulae for multiple automata simultaneously when the formulae are combined into a single formula which we can decide

its satisfiability for. The following paragraphs show how we use these features of Parikh images to determine satisfiability of multiple Parikh image formulae.

We can compute φ_1 for A_1 and φ_2 for A_2. Each φ_i represents exactly one A_i. Therefore, each φ_i by itself is satisfiable for A_i where φ_i describes words accepted by A_i[8].

If each φ_i is satisfiable, we want to know whether a combination of state language abstractions is compatible at the same time: $sat(\Phi^{PI}(p))$ such that $p = [q_1, q_2]$ is a product state and

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

However, to maintain the languages of specific automata distinguishable, we label each variable u_q, y_t (optionally, z_q, too) for each φ_i according to i: u_{iq}, y_{it} (z_{iq}). The only exception are free variables $\#_a$ which in contrary are bound to transition symbols $a \in \Sigma$ common to both A_i.

$sat(\Phi^{PI}(p))$ means there are words accepted by all φ_i simultaneously and therefore by both A_i. Consequently, the automata product would be non-empty.

3.3.2 Optimization Algorithm Using Parikh Images

We introduce the basic algorithm using Parikh image computation to construct the product of the intersection of finite automata. The algorithm resembles the length optimization algorithm from Algorithm 2. However, we compute Parikh image formulae and determine their satisfiability instead of generating lasso automata and determining satisfiability of length abstraction formulae now to optimize product construction.

We use Parikh image formulae to determine whether p is to be added to the generated product P (in case $sat(\Phi^{PI}(p))$) or omitted (in case φ_1 and φ_2 are unsatisfiable simultaneously in $\Phi^{PI}(p)$.

We can see our proposed algorithm using Parikh image computation to optimize product construction in the Algorithm 11. Similarly to the length abstraction algorithm, we start with the initial states (our abstract initial state, as described in Section 3.3.1) of A_1 and A_2, compute φ_1 and φ_2 combined into a single formula $\Phi^{PI}(p)$. If $\neg sat(\Phi^{PI}(p))$, P is empty and we can stop the product generation at once. Otherwise, $sat(\Phi^{PI}(p))$ is satisfiable and the corresponding product state is added to P. We proceed to generate the consecutive potential product states. We set the initial states for Parikh image formulae computation to the current state for each automaton A_i for each potential product state and recompute the combined Parikh image formula. We iterate over potential product states from W (see line 7).

The expression in line 9 computes state language abstractions by computing Parikh image formulae, determines their compatibility (satisfiability of Parikh image formulae) and returns the result as a boolean value. 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 result of the satisfiability test is used further in the algorithm to determine whether the product state is added to the generated product and consecutive potential product states are appended to W. The Parikh image is computed as it is explained in Section 3.3.1.

[8] Our Parikh image is an over-approximation of the accepted language of A_i. Therefore, there could exist such evaluation of variables in φ_i which describes words not accepted by A_i. It is a trade-off of precise representation of A_i for faster computation of φ_i.

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 $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

```
1  Q, δ, F ← ∅
2  I ← I_1 × I_2
3  W ← I
4  res ← False
5  solved ← ∅
6  while W ≠ ∅ do
7      picklast [q_1, q_2] from W
8      add [q_1, q_2] to solved
9      res ← α^PI(q_1) ∧ α^PI(q_2) is sat
10     if res = True then
11         add [q_1, q_2] to Q
12         if q_1 ∈ F_1 and q_2 ∈ F_2 then
13             add [q_1, q_2] to F
14         forall a ∈ Σ do
15             forall q_1' ∈ δ_1(q_1, a), q_2' ∈ δ_2(q_2, a) do
16                 if [q_1', q_2'] ∉ solved and [q_1', q_2'] ∉ W then
17                     add [q_1', q_2'] to W
18                 add [q_1', q_2'] to δ([q_1, q_2], a)
```

Algorithm 7: Product construction algorithm with Parikh image abstraction.

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 $p \to ap'$ 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.

```
1  if not isSkippable([q_1, q_2]) then
2      res ← α^PI(q_1) ∧ α^PI(q_2) is sat
3  else
4      res ← True
```

Algorithm 8: Parikh image computation with skippable states optimization.

3.3.3 Optimization with Incremental SMT Solving

We have to recompute Parikh image formula for every potential product state whose state language abstractions compatibility we check. We would appreciate a solution which would allow us to recompute only the conjuncts which change between two formulae (for two distinct product states) and keep the conjuncts which remain unchanged from the previous computation to be used in the next computation without the need to recompute them again. Our reduced Parikh image algorithm is designed for such optimization.

Notice that some conjuncts of Parikh image remain unchanged for the whole automaton, i.e., for every potential product state we compute Parikh images for. Therefore, we can use incremental solving features of SMT solver, which precompute these conjuncts only once

when Parikh image is first computed[9]. We make use of these already computed constraints to quicken Parikh image computation for every other state.

Assume finite automata A and B (whose intersection we generate) and a state $p = [a, b]$ where $a \in Q_A, b \in Q_B$ as a potential product state. The changes of conjuncts in φ_A and φ_B are caused by moving (setting) the states in both A and B corresponding to p as new initial states $I_A = \{a\}$ and $I_B = \{b\}$ as we proceed further into the automata in product construction. We start with the abstract initial states (one for each original automata, $I_A = \{a'_0\}$ and $I_B = \{b'_0\}$).

First, we compute $\Phi^{PI}(p_0)$ such that $p_0 = (a'_0, b'_0)$. Iff $sat(\Phi^{PI}(p_0))$, we generate new potential product states (e.g., $p_1 = (a_1, b_1)$ and $p_2 = (a_1, b_2)$). 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 $I_A = \{a_1\}, I_B = \{b_1\}$. Similarly for p_2, we would set $I_A = \{a_1\}, I_B = \{b_2\}$.

We now need to change every mention of initial states in φ_A and φ_B because the initial states are different from those we used at the start (a'_0 and b'_0) and for which we already computed $\Phi^{PI}(p_0)$. We now introduce an optimization of Parikh image computation which 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 and used throughout the whole process of working with the given finite automaton. Persistent clause consists of unchanged conjuncts of original Parikh image described in 3.3.1: conjuncts 2, conjuncts 3 and conjuncts 4.

State specific clause consists of conjuncts which change with every potential product state p tested for satisfiability, and as such have to be constructed and recomputed for every satisfiability test. The whole process of recomputing state specific clause is the most resource heavy part of the Parikh image computation algorithm. 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 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.

SMT solvers are well optimized to improve their performance by allowing incremental SMT solving. As we can see, the majority of conjuncts can be precomputed for the whole product generation and only taken into consideration with always recomputed new state specific clause.

It is worth to note that the conjuncts 3 manipulate with initial states but the structure of the conjuncts could be reversed to compute connectedness of the automaton in *reversed* order, from the accept states to the initial states. In that case, the conjuncts could be reconstructed as a part of the persistent clause dependent on accept states which remain unchanged (the abstract accept state) for the entire time. This additional optimization

[9]Consequently, computing Parikh image for the first time (for the first state of the given finite automaton) will take longer than for the following product states.

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.

SMT solvers can utilize their cache abilities to compute similar, consecutive formulae faster. We can observe how Parikh image satisfiability of successive product states are computed quickly due to minimal changes in formulae which SMT solvers can quickly resolve while using the most of the previously computed formulae constraints.

Algorithm for Incremental SMT solving Using Parikh Image

To implement incremental SMT solving to our current Parikh image computation shown in Algorithm 11, we need to make the following adjustments.

We need to precompute persistent clauses once A_1 and A_2. We insert a new line to our algorithm between lines 5 and 7. 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.

```
1 smtSolverPush()
2 res ← α^PI(q1) ∧ α^PI(q2) is sat
3 smtSolverPop()
```

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

The line 16 computes Parikh image formulae and determines their satisfiability, as explained in Section 3.3.1.

3.3.4 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 quicken 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 $sat(\Phi^{PI}(p))$.

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.

There is a problem with choosing the right time limit for timeout for SMT solver. We say the ideal timeout depends on a structure of finite automata we are working with, their size and complexity. And on how much time we are willing to give to the SMT solver. The timeout is directly proportional to the results precision and reversely proportional to the

scale of over abstraction computed. The cost is that the computation time requirements are directly proportional to SMT timeout, too.

3.4 Combination of State Language Abstractions

One of the strengths of our optimization algorithms is their high customizability. The different approaches can be combined, easily parallelized and applied on various operations on finite automata. We present an approach which takes advantage of specific strengths of our proposed optimization methods while trying to mitigate their weaknesses and utilizes them in a single algorithm.

We introduce a variation of satisfiability testing of state abstractions. We use both length abstraction and Parikh image computation to determine satisfiability of state abstraction to optimize Parikh image computation algorithm. The Algorithm 10 shows how we apply our optimizations on a single potential product state.

```
1 if α^LA(q1) ∧ α^LA(q2) is unsat then
2 |   res ← False
3 else
4 |   res ← α^PI(q1) ∧ α^PI(q2) is sat
5 |   if res = Unknown then
6 |   |   res ← True
```

Algorithm 10: Implementation of function checking satisfiability of state abstraction 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 Parikh image algorithm 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 11).

3.5 Abstraction of State Language with Mintermization

In this section, we introduce a method of optimizing operations on finite automata using minterms of the given finite automata. Minterm computation abstracts state language of automata in a different approach than which we 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 to compute minterms for the non-empty multiset of input finite automata $A = M_1, M_2, \ldots, M_n$, 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, but might create a more concise finite automata which will be easier to work with 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. Before all else, let us explain what minterms are and how you can generate them.

Definition 3.5.1 (Minterms) *Given an NFA* $M = (Q, \Sigma, \delta, I, F)$*, let* $\Phi = \{\varphi_1\varphi_2, \ldots, \varphi_n\}$ *be a finite set of non-empty finite sets of transition symbols* $\varphi_i = \{a | a \in \Sigma \wedge q \xrightarrow{a} q'\}$*,* $1 \leq i \leq n$*,* $q, q' \in Q$ *where* 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 the given NFA M such that

$$Minterms(\Phi) = \left\{ \psi = \bigcap_{1 \leq i \leq n} \psi_i \middle| \forall i \in \{1, \ldots, n\}.(\psi_i \in \{\varphi, Q \setminus \varphi\}) \wedge \psi \neq \varnothing \right\}.$$

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 NFA, which allows us to use minterms state language abstraction for optimization of operation on those automata.

Let us consider finite automata $M_1 = (\{s_0, s_1, s_2, s_3\}, \Sigma, \delta_1, \{s_0\}, \{s_3\})$ and $M_2 = (\{q_0, q_1, q_2\}, \Sigma, \delta_2, \{q_1\}$ over alphabet $\Sigma = \{a, b, c, d\}$ with δ_1 and δ_2 according to Figure 3.9 and Figure 3.10, respectively.

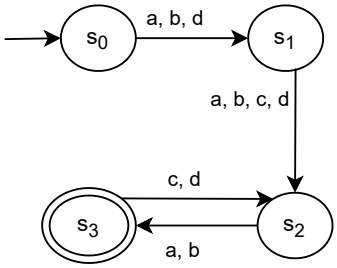

Figure 3.9: Finite automaton M_1 with transitions δ_1.

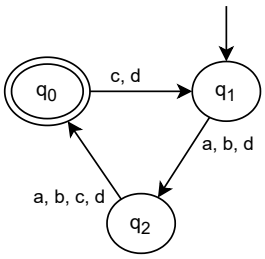

Figure 3.10: Finite automaton M_2 with transitions δ_2.

Figure 3.11: Finite automata M_1 and M_2 used as example automata for mintermization.

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.

If we were to compute minterms for these automata, we would proceed as follows: Starting with the whole alphabet of both automata[10] at the top of the minterm tree to

[10] If the automata had non-equal alphabets, we would start with their intersection: $\Sigma = \Sigma_1 \cap \Sigma_2$

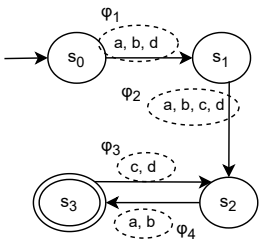

Figure 3.12: Finite automaton M_1 with transition sets φ_i.

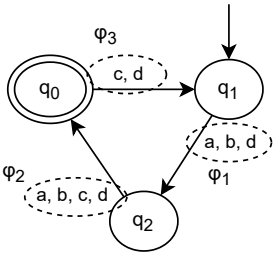

Figure 3.13: Finite automaton M_2 with transition sets φ_i.

Figure 3.14: Finite automata M_1 and M_2 with marked transition sets used in mintermization.

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.

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 Figure 3.15 illustrates our mintermization process in a diagram.

The acquired minterms are:

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

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. The resulting automata can be seen in Figure 3.18.

Consequently, assuming the previously said, 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. We make use of this knowledge further.

In the following section, we propose a method of using minterm computation with Parikh image computation optimization. We choose this approach in order to mitigate the disadvantages of Parikh image computation for finite automata, especially those 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 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 for every single transition symbol between two states. We can now compute possibly fewer transitions with the resulting minterms instead.

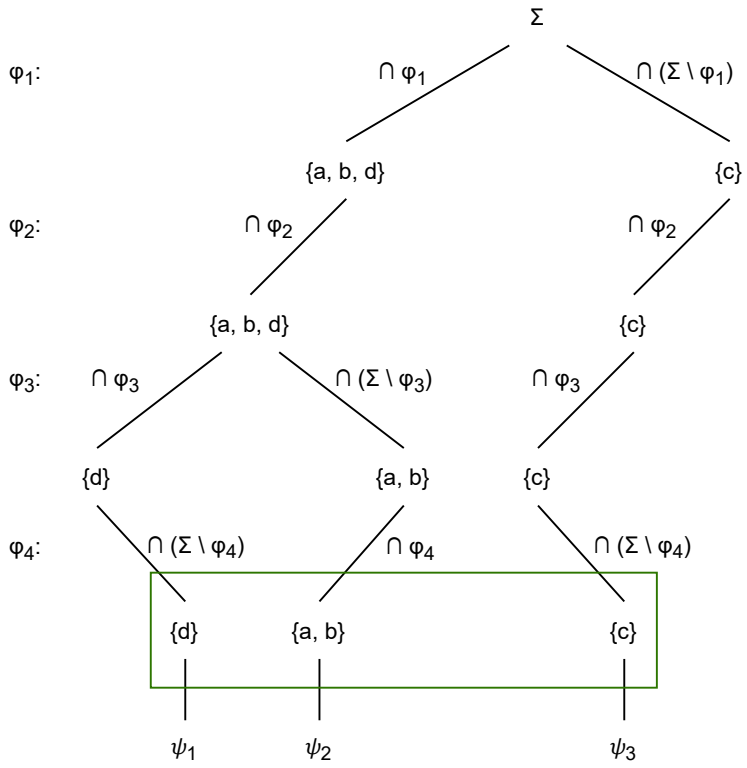

Figure 3.15: Mintermization process executed on example finite automata M_1 and M_2. We start with the whole alphabet and make our way down through all mintermization sets φ_i, where $1 \leq i \leq n$. For each mintermization set, we compute the intersection of the preceding set with the current mintermization 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.

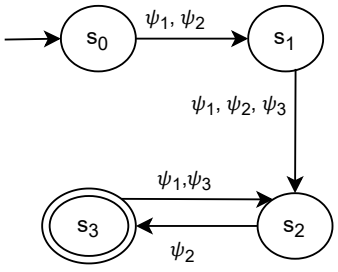

Figure 3.16: Finite automaton M_1 with transitions substituted by corresponding minterms $psi_i \in \Psi$ created from these transition sets.

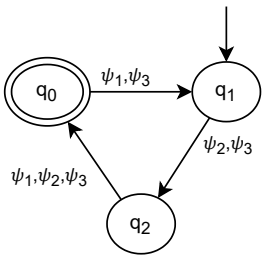

Figure 3.17: Finite automaton M_2 with transitions substituted by corresponding minterms $psi_i \in \Psi$ created from these transition sets.

Figure 3.18: Finite automata M_1 and M_2 with substituted transitions with minterms in the process of mintermization.

Chapter 4

Experiments and Results

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.

Test benchmarks used in our experiments were obtained from regular model checking. We have tested various different finite automata and their combinations. We have often used the same automata with their slightly changed variations to simulate real world examples of usually used automata to see how the optimized algorithm reduces the generated state space for certain types of automata with their typical qualities.

We have tested two main aspects:

- First, we have tested the generated state space for emptiness test. That is, whenever we find a solution—accepting state in the intersection, the test ends, and we count the number of generated product states to this moment. If no intersection is found, we end the test when it is certain there is no accepting state and the intersection is indeed empty.
- Second, for the same pair of automata, we have tested the full product construction. Adding new accepting states along the way and comparing generated state spaces in the end for the full product accepting the whole intersection of original automata.

4.1 Length Abstraction

The following graphs show the results for both the emptiness test and full product construction. The graph in Figure 4.1 shows the comparison of product state spaces sizes in basic product construction algorithm and our optimized algorithm considering length abstraction for emptiness test. Sorted in order of increasing product state space size generated by the basic product construction algorithm. The graph in Figure 4.2 shows the same data, only for the full product construction experiment.

Where the length abstraction cannot optimize the product construction, both products have about the same state space size. These results are caused mostly by constructing products of two almost identical automata with only a few states/transitions missing/added

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

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

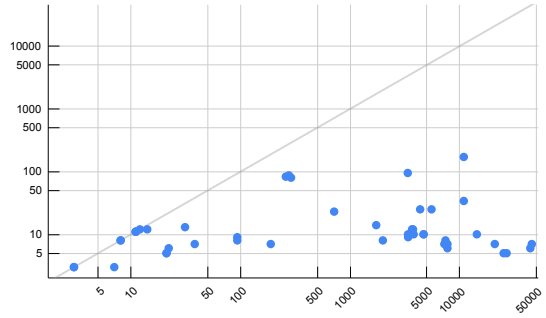

Figure 4.1: Emptiness test

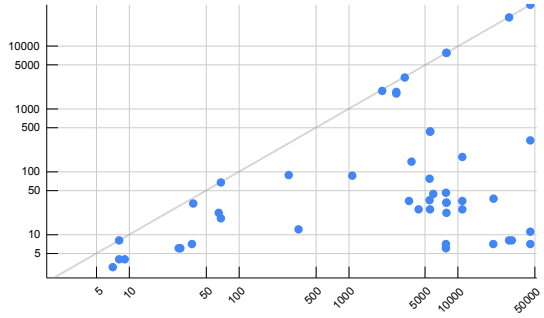

Figure 4.2: Full product construction

Figure 4.3: Comparison of state space sizes generated by basic and optimized product construction algorithms. Both axes are in logarithmic scale, x-axis showing state space sizes of basic product, y-axis state space sizes of optimized product.

which do not affect the accepting runs for recognized languages. There are therefore no branches which can be trimmed—most of the processed states are evaluated as *satisfiable* in length abstraction satisfiability check. In full product construction results, if there are nearly no product states to trim, the generated product state space size *explodes* similarly to the basic product construction algorithm—typical for automata with large numbers of transitions from every state causing large numbers of possible accepted lengths, where our algorithm can trim only a few states.

For another automata, the product generated by our algorithm is much smaller. We can see from the graphs that the larger the basic product state space size gets, the higher impact our optimization has on the product state space size. The same holds for the full product construction results. For cases where the intersection is truly empty and accepted lengths differ in both automata, our algorithm stops the process of product construction on the very first tested product state. The basic algorithm continues to create a full product.

We get the best results for automata with practically the same transitions which differ only slightly in final states or a few transitions which affects the accepting runs in the original automata. These changes cause the basic algorithm to generate the product states without realizing most (if not all) product states do not lead to an accepting state. These slight differences in automata (especially in final states) usually also change the length of accepted words. Therefore, our optimization is able to notice these differences and trim most of the product state space, if not the whole product, when no final state can be accessed and the intersection is empty.

In both graphs, we can see the aforementioned quadratic state space explosion for product is nearly not affecting our algorithm in comparison to the basic product construction algorithm. Optimized products are easier to work with and operations on such products require less computational time and memory consumption.

It is worth mentioning that we have neglected the number of generated state space for our lasso automata this whole time. We can use deterministic minimization on original automata to further optimize the generated state space for lasso automata and products. However, we do not need these lasso automata after product construction is complete. Therefore, the lasso automata do not affect how efficiently we work with the generated

product. Nevertheless, the number of generated lasso states in the process of deciding the intersection emptiness test matters. For different automata, the generated state space varies. For state space sizes of lasso automata in our experiments, see Codeberg repository[3].

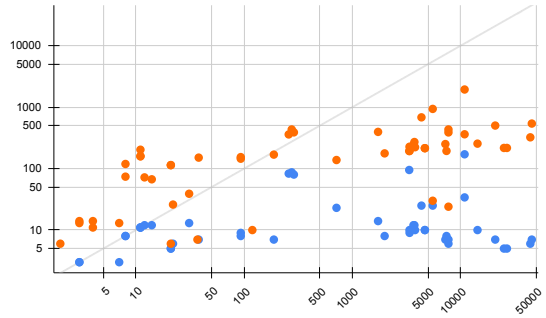

Figure 4.4: Emptiness test

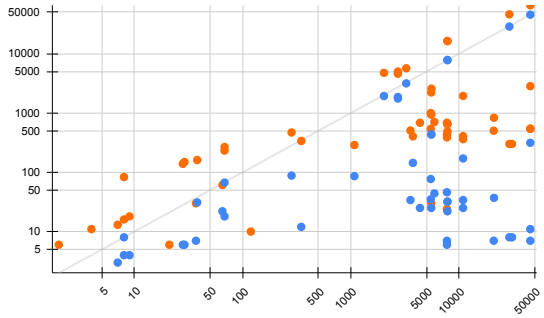

Figure 4.5: Full product construction

Figure 4.6: Comparison of state space sizes generated by basic and optimized product construction algorithms with sum of states generated for both the final optimized product and lasso automata states generated in the process of the product construction. Both axes are in logarithmic scale, x-axis showing state space sizes of basic product, y-axis state space sizes of optimized product (and optimized product with lasso automata states). The blue dots represent only the optimized product state space sizes (as in Figure 4.3), the orange dots the sum of optimized product state space sizes and the generated lasso states.

As we can see in Figure 4.6, even when counting with lasso automata product state space sizes, 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, whose intersection is computed, the expense of generating lasso automata is significant in comparison with the generated product state space sizes. The larger the original automata get, the lesser the expense of the number of lasso automata states is in comparison with the basic product state space.

Out of all experiments, one weakness of our algorithm is clear—the more final states the original automata have, the more difficult it is to optimize the full product construction using length abstraction. This is caused by the fact that every final state increases the number of accepted different lengths per automaton. Therefore, with automata where out of hundreds or thousands of states nearly every state is a final state too, our optimization algorithm has to consider multiple possible lengths and cannot easily determine which branches will not be accepted by the product automaton.

4.2 Parikh Image Computation

In this section, we show results of several experiments with Parikh image computation optimization. At first, we are interested in pruning capabilities of Parikh image abstraction without further optimizations. Later, we provide results for introduced optimizations of Parikh image computation algorithm.

[3] https://codeberg.org/Adda/optifa/src/branch/master/results

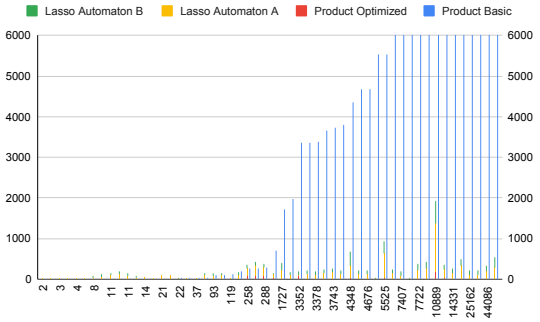

Figure 4.7: Emptiness test

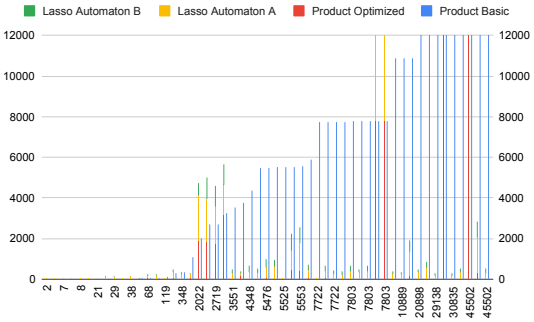

Figure 4.8: Full product construction

Figure 4.9: Stacked comparison of state space sizes generated by basic and optimized product construction algorithms with sum of states generated for both the final optimized product and lasso automata states generated in the process of the product construction. Both axes are in logarithmic scale, x-axis showing state space sizes of basic product (ordered in ascending order), y-axis state space sizes of depicted experiments—because of huge differences in sizes of basic product and optimized product with lasso automata, the largest shown values are set to 6000 and 12000, respectively. Each two columns show a single experiment with our optimized solution as the left (green, red and orange) column—as a sum of all generated states (of optimized product (green) and both lasso automata (red and orange), and the right blue column as the basic product state space size).

The following graphs show the results for both the emptiness test and full product construction of unoptimized Parikh image computation abstraction. The graph in Figure 4.13 shows the comparison of product state spaces sizes in basic product construction algorithm and our Parikh image computation algorithm for emptiness test. Sorted in order of increasing product state space size generated by the basic product construction algorithm. The graph in Figure 4.14 shows the same data, only for the full product construction experiment.

We conclude from the experiments that Parikh image optimizes the generated product state space in nearly every case. The strength of Parikh image is its higher pruning capacity due to wider range of information gathered from the automata. In multiple cases, Parikh image optimization is able to prune vast *branches* of potential generated product by correctly determining incompatible transition symbols even if possible lengths of accepted words are mutually compatible.

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

We can notice that the number of persistent clauses covers substantial part of all Parikh image clauses (experimentally determined to be usually around 70% for our benchmark automata). Therefore, around 70% of each computed Parikh image clauses can be precomputed once. Only 30% of clauses must be computed repeatedly for each potential product state.

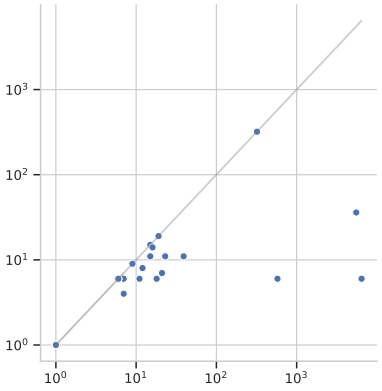

Figure 4.10: Emptiness test

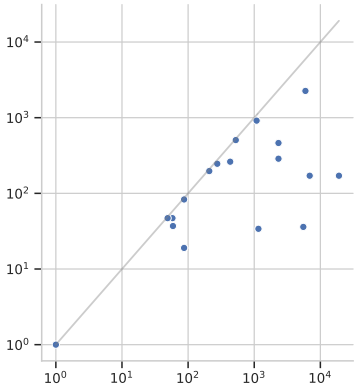

Figure 4.11: Full product construction

Figure 4.12: Comparison of state space sizes generated by basic and optimized with Parikh image computation product construction algorithms. Both axes are in logarithmic scale, x-axis showing state space sizes of basic product, y-axis state space sizes of optimized product.

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

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

4.3 Experiments with Combination of State Language Abstractions

We show results of several experiments with Parikh image computation optimization. At first, we are interested in pruning capabilities of Parikh image abstraction without further optimizations. Later, we provide results for introduced optimizations of Parikh image computation algorithm.

The following graphs in Figure 4.15 show the results for both the emptiness test and full product construction of unoptimized Parikh image computation abstraction. The graph in Figure 4.13 shows the comparison of product state spaces sizes in basic product construction algorithm and our Parikh image computation algorithm for emptiness test. Sorted in order of increasing product state space size generated by the basic product construction algorithm. The graph in Figure 4.14 shows the same data, only for the full product construction experiment.

We conclude from the experiments that Parikh image optimizes the generated product state space in nearly every case and produces equal or better results than length abstraction

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

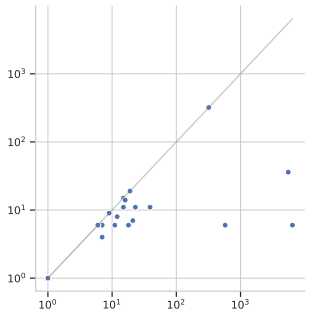

Figure 4.13: Emptiness test

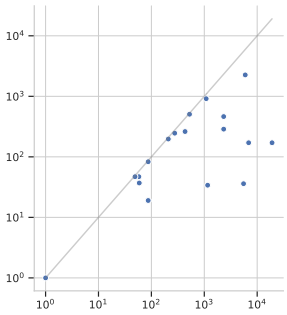

Figure 4.14: Full product

Figure 4.15: Comparison of state space sizes generated by basic and optimized product construction algorithms with length abstraction (blue dots) and Parikh image computation (orange dots). Both axes are in symmetrical logarithmic scale[5], showing state space sizes: x-axis of basic product, y-axis of optimized product.

every time. The strength of Parikh image is its higher pruning capacity due to wider range of information gathered from the automata. In multiple cases, Parikh image optimization is able to prune vast *branches* of potential generated product by correctly determining incompatible transition symbols even if possible lengths of accepted words are mutually compatible, sometimes even entirely stopping product construction immediately when basic and length abstraction constructions continue to generate state space further.

Incremental SMT solving proves to be a great improvement to the Parikh image computation optimization. The amount of clauses depends on the number of states in finite automata, the number of transitions and the number of initial or accepting states. The following experiment provides an example comparison of the number of all clauses in Parikh image, clauses common to all product states (persistent clauses) and state specific clauses. For a product of 434 states, each product state Parikh image contains 2652 clauses. From those, 1782 clauses are persistent clauses and the remaining 870 are state specific clauses. A proportional ratio o persistent clauses in whole Parikh image is around 67.2%. The number of persistent clauses (experimentally determined to be usually around 70%) for our benchmark automata means around 70% of each computed Parikh image clauses can be precomputed once. Only 30% of clauses must be computed repeatedly for each potential product state.

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 accepting state—that is, omitting the *branches* which do not lead to any accepting state—by performing the emptiness test of such states using various state languages abstractions over the original automata 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 is minimized especially for intersections with huge non-terminating branches or for intersections of automata accepting different lengths of words recognized by the automata languages. Further, for automata with long lines and similar automata varying only slightly from each other. Experiments show our algorithm generates smaller product state spaces for both emptiness test and full product construction, which are two usually used operations on automata intersection. All our abstractions consider over-approximation of possible products. Therefore, our optimizations are safe to use for any uses resolving operations on finite automata.

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 [4] and predicate abstraction [5, 12], IMPACT [17], possibly IC3/PDR [13, 3]. 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 [14, 20, 6] and on the interpolation-based approach of McMillan [2, 11].

Bibliography

[1] Abdulla, P. A., Atig, M. F., Chen, Y., Holík, L., Rezine, A. et al. String Constraints for Verification. In: *CAV*. Springer, 2014, vol. 8559, p. 150–166. Lecture Notes in Computer Science.

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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

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 $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

```
1  Q, δ, F ← ∅
2  I ← I_1 × I_2
3  W ← I
4  res ← False
5  solved ← ∅
6  addPersistentClauses()
7  while W ≠ ∅ do
8      picklast [q_1, q_2] from W
9      add [q_1, q_2] to solved
10     if not isSkippable([q_1, q_2]) then
11         if α^LA(q_1) ∧ α^LA(q_2) is unsat then
12             res ← False
13         else
14             smtSolverPush()
15             addStateSpecificClauses([q_1, q_2])
16             res ← α^PI(q_1) ∧ α^PI(q_2) is sat
17             smtSolverPop()
18             if res = Unknown then
19                 res ← True
20     else
21         res ← True
22     if res = True then
23         add [q_1, q_2] to Q
24         if q_1 ∈ F_1 and q_2 ∈ F_2 then
25             add [q_1, q_2] to F
26         forall a ∈ Σ do
27             forall q_1' ∈ δ_1(q_1, a), q_2' ∈ δ_2(q_2, a) do
28                 if [q_1', q_2'] ∉ solved and [q_1', q_2'] ∉ W then
29                     add [q_1', q_2'] to W
30                 add [q_1', q_2'] to δ([q_1, q_2], a)
```

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