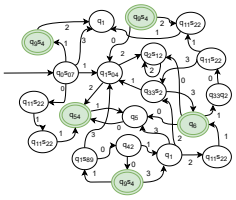
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

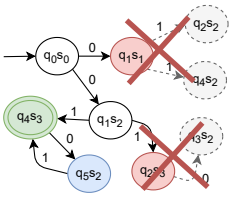
Excel@FIT 2022
http://excel.fit.vutbr.cz

Abstraction of State Languages in Automata Algorithms

David Chocholatý*



Are the operations on finite automata
too complex?
Do not despair, we have a solution.



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.

Keywords: Finite Automata — State Languages Abstraction — SMT solving — Product Construction — Emptiness Test — Intersection Computation Optimization — State Space Reduction — Length Abstraction — Parikh Image

Supplementary Material: Optimization Code and Experiments Results — Code Symboliclib

*xchoch08@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Automata theory is a well-known field of study used in many areas. Automata are commonly used in mathematics and computation theory in general (e.g., in model checking [1] or string solving and analysis [2]). Their usage in the field of logic is just as important, too (e.g., WS1S [3]).

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). Our focus is on 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.

The intersection of finite automata 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

[1]The product construction sometimes *explodes* in a huge prod-

parts of the generated state space which cannot accept any words (no final states can 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.

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 accepting states and do not continue from such states. When states language abstraction 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.

Our goal is to explore possibilities of using various abstractions of automata state languages[2] in optimization of automata algorithms. We consider options of using them to pruning pairs of states with incompatible abstractions. We continue towards optimization of these techniques. Our suggested optimization methods are applicable on many other typical automata algorithms. Consequently, our discoveries have wider impact on multiple automata operations.

We have previously used length abstraction of state languages[3] optimization in [4]. 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 not ideal. 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.

The optimization approach we consider now is the computation of Parikh images[4] for product states. Parikh image of a word tells us how many times a symbol occurs in a word[5]. 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. 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[6], 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 to be empty, which can be quickly determined by our optimization, whereas the classic unoptimized algorithms would proceed to generate useless fragments of suppositional product.

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, this optimization process works really well. Parikh image abstraction usually trims vast state spaces where length abstraction cannot prune anything and basic product state space explodes exponentially (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. heuristic trimming generated state space of operations on finite automata based on Parikh image computation, and
2. implementation and experimental evaluation of said heuristic and its optimizations.

2. Preliminaries

Let us clarify a few definitions and terms often used throughout this paper. The following definitions are mostly adapted from [5] or [6].

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

uct state space.

[2] Over-abstractions of original state languages, precisely. Therefore, we cannot accidentally trim product states leading to at least one accept state.

[3] to sets of possible word lengths

[4] represented as semi-linear sets

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

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

alphabet is a *word* w from the set of all words Σ^* over an alphabet Σ.

Definition 2.1 (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 states****.*

A *run* of A on input $a_0 a_1 a_2 \dots a_{n-1}$ is a sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_n$, such that $q_i \in Q$ for $0 \le i \le n$, $q_0 = I$ and $\delta(q_i, a_i) = q_{i+1}$ for $0 \le i \le n-1$. A run is *accepting* if $q_n \in F$. 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 accepting states (or non-terminating states[7] in case the word is not accepted by the automaton at all).

Definition 2.2 (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****:* $\delta : Q \times \Sigma_\varepsilon \to P(Q)$*, where* $\Sigma_\varepsilon = \Sigma \cup \varepsilon$ *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****.*

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 2.3 (Product construction) *Operations on automata* A_1 *and* A_2 *yield a result – a product A as a 5-tuple deterministic finite automaton* $A = (Q, \Sigma, \delta, I, F)$*.*

[7] No accept state is accessible from them.

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* $F = F_1 \times F_2$*.*

The transition relation 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.

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 ← I1 × I2
3  W ← I
4  while W ≠ ∅ do
5      pick [q1, q2] from W
6      add [q1, q2] to Q
7      if q1 ∈ F1 and q2 ∈ F2 then
8          add [q1, q2] to F
9      forall a ∈ Σ do
10         forall q1' ∈ δ1(q1, a), q2' ∈ δ2(q2, a) do
11             if [q1', q2'] ∉ Q then
12                 add [q1', q2'] to W
13             add [q1', q2'] to δ([q1, q2], a)
```

Algorithm 1: Classic product construction

Definition 2.4 (Galois Connection) *Galois connection is a quadruple* $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ *such that:*

- $\mathcal{P} = \langle P, \le \rangle$ *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* $\forall q \in Q$*:*

$$p \le \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q.$$

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, $\forall p \in P : p \le \gamma(\alpha(p))$. That is, the abstraction may only over-approximate the concrete semantics.

3. State Language Abstractions

We try to optimize operation on automata with our optimizations methods. For the purpose of introducing our methods, we focus solely on synchronous product construction of automata intersection in this paper. However, the proposed optimizations can be applied to other operations as well. When constructing a product, a considerate number of generated product states are nonterminating 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. We can thus prune nonterminating 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 their compatibility.

Our optimization are applicable on two and more automata, but for the ease of explanation, we will consider only two automata (A_1 and A_2). We have introduced the concept of state language abstractions in [4] with length abstraction of state languages. The idea is to find a state language abstraction $\alpha^X(q)$ of a state q in abstraction X ($\alpha^{LA}(q)$ for length abstraction and now $\alpha^{PI}(q)$ 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 states.

The $\alpha^{LA}(q)$ is a fast and simple abstraction abstracting accepted words to only their lengths, but can be too general to detect nonterminating states in some cases. In this section, we present a product construction optimization using Parikh image state language abstraction which tries 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 without consideration of which transition symbols are actually used. Parikh image abstraction allows us to more precisely determine whether $A_1 \cap A_2 = \varnothing$. 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.

3.1 Parikh Image

We derive our Parikh image construction from the Parikh's theorem [7] described in [8], creating a semilinear 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 [9] 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.

3.1.1 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.

3.1.2 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].

[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.

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,

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

$sat(\psi)$ is *True* iff ψ is satisfiable, *False* otherwise.

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.

3.2 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 [4]. 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 2. Similarly to the length abstraction algorithm, we start with the initial states (our abstract initial state, as described in Section 3.1.1) 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 6).

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 3.1.1.

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

1 $Q, \delta, F \leftarrow \varnothing$
2 $I \leftarrow I_1 \times I_2$
3 $W \leftarrow I$
4 $res \leftarrow False$
5 $solved \leftarrow \varnothing$
6 **while** $W \neq \varnothing$ **do**
7 **picklast** $[q_1, q_2]$ **from** W
8 **add** $[q_1, q_2]$ **to** $solved$
9 $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat
10 **if** $res = True$ **then**
11 **add** $[q_1, q_2]$ **to** Q
12 **if** $q_1 \in F_1$ ***and*** $q_2 \in F_2$ **then**
13 **add** $[q_1, q_2]$ **to** F
14 **forall** $a \in \Sigma$ **do**
15 **forall** $q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)$ **do**
16 **if** $[q_1', q_2'] \notin solved$ ***and*** $[q_1', q_2'] \notin W$ **then**
17 **add** $[q_1', q_2']$ **to** W
18 **add** $[q_1', q_2']$ **to** $\delta([q_1, q_2], a)$

Algorithm 2: Product construction algorithm with Parikh image abstraction.

3.2.1 Optimization with Skippable States

Same as for the length abstraction algorithm from [4], we can make use of skipping satisfiable product states optimization. When $sat(\Phi^{PI}(p))$ for some potential product state $q = [q_1, q_2]$ and q generates only one consecutive potential product state $q' = [q_1', q_2']$ such that $q \rightarrow aq'$ where $a \in \Sigma$, we can skip computing Parikh images for q' as we know for sure $sat(\Phi^{PI}(p'))$ in order to get a satisfiable result for Parikh image for state q. We can add this functionality to our previous algorithm by replacing line 9 with the content of Algorithm 3.

1 **if** ***not*** `isSkippable`$([q_1, q_2])$ **then**
2 $res \leftarrow \alpha^{PI}(q_1) \wedge \alpha^{PI}(q_2)$ **is** sat
3 **else**
4 $res \leftarrow True$

Algorithm 3: Parikh image computation with skippable states optimization.

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.

3.3.1 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.

[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.

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 3.1: 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 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.

3.3.2 Algorithm for Incremental SMT solving Using Parikh Image

To implement incremental SMT solving to our current Parikh image computation shown in Algorithm 2, 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 6. The new line contains a call to a function `addPersistentClauses()` which precomputes persistent clauses for both A_1 and A_2. Note that the function is called only once, before we enter the *while* loop for iterating over potential product states.

We compute state specific clauses as normal when we ask whether $sat(\Phi^{PI}(p))$ when we are checking compatibility of both α^{PI} on line 9. However, we push the previously precomputed state persistent clauses to the SMT solver stack. This preserves them when the current state specific clauses are dropped after $sat(\Phi^{PI}(p))$ is resolved. For a pseudocode of the replacement of line 9, see Algorithm 4.

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

Algorithm 4: Add state specific clauses to SMT solver for incremental SMT solving optimization.

The line 2 computes Parikh image formulae and determines their satisfiability, as explained in Section 3.1.1.

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 re-

sults 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.

3.5 Parikh Image Optimization Enhanced by Length Abstraction

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 5 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 5: 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 2).

4. Experiments and Results

The reference implementation[10] 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[11]. 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.

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 3 show the results for both the emptiness test and full product construction of unoptimized Parikh image computation abstraction. The graph in Figure 1 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 2 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 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

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

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

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

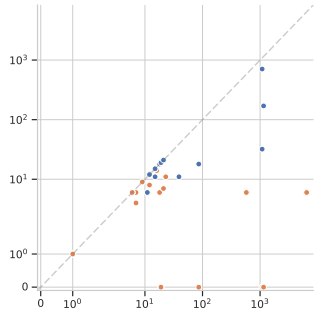


Figure 1. Emptiness test

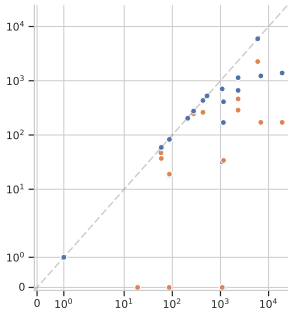


Figure 2. Full product

Figure 3. 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[12], showing state space sizes: x-axis of basic product, y-axis of optimized product.

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.

5. Conclusions

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 large branches where no final states can be reached 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 [10] and predicate abstraction [11], IMPACT [12], possibly IC3/PDR [13]. 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] and on the interpolation-based approach of McMillan [15].

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.

References

[1] Stephen F. Siegel and Yihao Yan. Action-based model checking: Logic, automata, and reduction. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 77–100. Springer, 2020.

[2] Anthony Widjaja Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL*, pages 123–136. ACM, 2016.

[3] Tomás Fiedor, Lukás Holík, Petr Janku, Ondrej Lengál, and Tomás Vojnar. Lazy automata techniques for WS1S. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 407–425, 2017.

[4] David Chocholatý and Lukáš Holík. Optimizing automata product construction and emptiness test. In *Excel@FIT 2021*, 2021. https://codeberg.org/Adda/excel_at_fit_2021/src/branch/master/paper.pdf.

[5] Javier Esparza. Automata theory: An algorithmic approach. online, 2017. https://www7.in.tum.de/~esparza/automatanotes.html.

[6] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[7] Dexter C. Kozen. Parikh's theorem. In *Automata and Computability*, pages 201–205, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

[8] Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. Parikh's theorem: A simple and direct automaton construction, 06 2011.

[9] Petr Janků and Lenka Turoňová. Solving string constraints with approximate parikh image. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory – EUROCAST 2019*, pages 491–498, Cham, 2020. Springer International Publishing.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[11] Michael Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998.

[12] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[13] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[14] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2(POPL):4:1–4:32, 2018.

[15] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 2013.