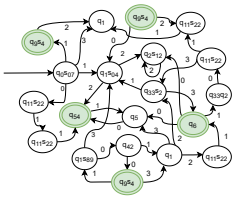
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

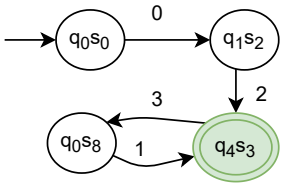
Excel@FIT 2021
http://excel.fit.vutbr.cz

Optimizing Automata Product Construction and Emptiness Test

David Chocholatý*, Lukáš Holík**



If you had to, which finite automaton would you rather work with?



Abstract
Finite automata theory is a well-known model of computational theory. We use them widely in many situations. We will focus our attention to different heuristics for optimizing several typical problems with finite automata. We are interested especially in computation of intersection of automata product construction and its emptiness test, which is required in modern computation technologies, but requires a lot of computational time and generates unnecessary state space. For this reason, we will try to use a length abstraction for solving these problems and optimizing the product construction and its emptiness test as good as possible using solely knowledge about recognized words lengths.

Keywords: Finite Automata — Product Construction — Emptiness Test — Intersection Computation Optimization — State Space Reduction — Length Abstraction

Supplementary Material: Optimization Code and Experiments — Code Symboliclib

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

1. Introduction

Automata theory is a well-known field of study. Hard to believe maybe, but the truth is automata appear in our mundane lives every day. It is consequently important to study automata further and learn to cooperate with them too. Automata are also commonly used in mathematics and computation theory in general (e.g., in model checking [1] or string solving and analysis [2]). Of course, their uses in the field of logic are not to be forgotten (e.g., WS1S [3, 4]).

Finite automata are easy to understand, but as soon as you start adding new states, transitions and additional accepting states, finite automaton will get extensively larger and harder to work with. One of such operations over the automata is construction of their intersection.

The intersection of two or more automata (the synchronous product construction) is often used in mathematics and logic. It is extensively used operation where one follows the transitions in the original automata trying to find such runs where all of the original automata use the same transition symbols. Therefore, one can do the same run with the transition symbols for all of the finite automata at once in the product—assuming there is such a run—combining the original states from the individual runs to tuples called product states and adding them to generated state space. For two automata, the classical algorithm starts with an initial state of both automata, make pairs from them and try to find transitions leading from these initial states in the original automata with the same symbol. Generate new pairs for found transitions, find their transitions and so on. Every product state represents an intersection of languages of two corresponding states in the

original automata.

Unfortunately, the synchronous product construction is extremely expensive on computational time as one needs to generate a vast amount of product states during the process. For two automata, the state space can increase quadratically, for more automata, the product state space increases exponentially according to the amount of used automata and the amount of their states. Furthermore, often there are large parts of the generated state space which cannot accept any words (non-terminating states), yet are still generated because of the corresponding transition symbols.

We will try to optimize this process by reducing the amount of generated product states and their transitions for both generating the synchronous product and deciding its emptiness test. We focus mainly on decision making about the satisfiability problem solving the emptiness of an intersection of two (eventually even multiple) finite automata. We consider length abstraction over the initial finite automata while trying to predict which product states cannot lead to any accepting states.

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 omit such states from our generated product. Consequently, this removes the need to even consider their potential successor states, which are generated normally. By doing this, we trim the generated product state 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.

Computing length abstraction over the languages of finite automata (and over individual states in the automata in particular) is accomplished using lasso automata (handle and loop automata)—deterministic finite automata with an unary alphabet (similar as in [5]). They consist of a *handle* (a sequence of states from the initial state) and a single *loop* (resolving the cycles in the original automaton) with a few accepting states along the way, hence the name of a handle and loop or lasso automaton. Create one by taking the initial automaton, consider all transition symbols as a single transition symbol and determinize this automaton.

What you get is an automaton accepting every length of any word recognized by the language of the initial automaton. Consequently, it is easy to compute semi-linear set (formulae[1]) for the allowed lengths of words, which can be effectively processed and compared using SMT solvers. We are computing these formulae for 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 test is resolved as satisfiable.

We have implemented this optimization 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 amount of trimmed product states in the process. For certain types of automata of certain qualities, this optimization process works really well.

Our proposed algorithm cannot remove any product states leading to an actual accepting state. Therefore, it is impossible to accidentally trim product states leading to some accepting states and change the intersection language in the end. The language we get is the same as the one generated from the basic product construction algorithm. Consequently, is it completely secure to use our optimization with any kind of automata for any kind of uses.

The contribution of this work can be summarized as follows:

1. heuristic trimming generated state space of finite automata synchronous product construction based on length abstraction, and
2. implementation and experimental evaluation of said heuristic.

2. Preliminaries

Let us clarify a few definitions and terms often used throughout this paper. The following definitions are mostly adapted from [6] or [7].

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

- Q *is a nonempty* ***set of states****,*
- Σ *is an* ***input alphabet****,*

[1] disjunction of linear equations

- 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 $a_0a_1a_2...a_{n-1}$ is a sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_3} ... \xrightarrow{a_{n-1}} q_n$, such that $q_i \in Q$ for $0 \leq i \leq n$ 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 $L(A) = \{w \in \Sigma^* | w \text{ is accepted by } A\}$. 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 transition symbol a*. For every state, DFA has at most one transition for a given symbol.

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****:* $\delta : Q \times \Sigma \to P(Q)$*, where* $P(Q)$ *is a set of subsets of* Q*, and*
- I *is a nonempty* ***set of initial states****.*

Consequently, DFA has exactly one run on a given word from initial state to one of the accepting states (or nonterminating states in case the word is not accepted by the automaton at all). For every state and its transition $P(Q) \in \delta(q,a)$ is a singleton.

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 DFA B. *Determinization* is a process of converting such NFA to DFA.

Definition 2.3 (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* $Q' = 2^Q$*,* $F' = \{S \in Q' | S \cap F \neq \emptyset\}$*,* $I' = I$ *and for* $S \in Q' : \delta'(S,a) = \bigcup_{s \in S} \delta(s,a)$.

Definition 2.4 (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* $F = F_1 \times F_2$.

The transition relation is described as $\delta = ([q_1, q_2], a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$. 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

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

To solve the *emptiness test*, we test

$$L(A) = \emptyset.$$

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

3. Product Construction Optimization with Length Abstraction

Our task is to try to minimize the amount 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[2]. 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, but first some rudimentary knowledge on length abstraction is needed.

3.1 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.

Length abstraction generalizes the language recognized by the initial automaton with considering only the possible lengths of words accepted by the automaton. It is an over-approximation of the language accepted by the original automaton. For us, this means if a word is not accepted by the length abstraction automaton, it cannot be accepted by the initial automaton either.

The length abstraction automaton is represented by a so-called lasso automaton. Let us demonstrate creation of the lasso automaton on the following simple non-deterministic finite automaton A_1, which we will continue to use in this paper to depict our optimization algorithm.

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

Transition relation δ_1 is depicted in Figure 1.

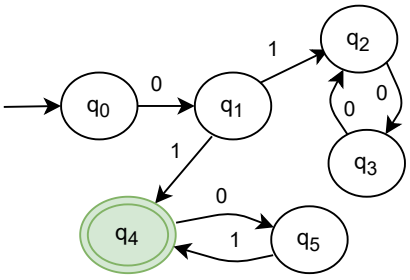


Figure 1. Non-deterministic finite automaton A_1

NFA A_1 is a non-deterministic finite automaton (see state q_1) and accepts more than one input symbol. Due to the fact that we work only with recognized word lengths, we can substitute automaton alphabet with unary alphabet of single input symbol (we have chosen $*$ for demonstration purposes)[3]. Then, we can compute lasso automaton for our original automaton A_1 with unary alphabet, which is its deterministic equivalent.

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

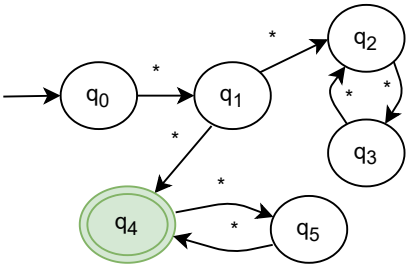


Figure 2. Non-deterministic finite automaton A_1 with unified transition symbols

We start the determinization process on our updated automaton. For the final lasso automaton A'_1 for the original automaton A_1, see Figure 3. This automaton now accepts any words of lengths of words recognized by the original automaton. We will use these lengths in the process of constructing the product.[4]

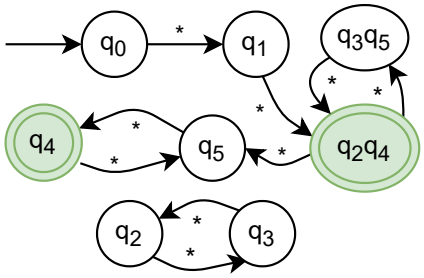


Figure 3. Lasso automaton A'_1 for the original NFA A_1

3.2 Single Lasso Automaton for Each Original Automaton

When we are constructing a product, we do not want to regenerate the lasso automaton L for each new product state q. This is inefficient. Therefore, our algorithm

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

[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 this lasso automaton looks different to what is depicted in Section 3.1. It is caused by our optimization with generating only one single lasso automaton per original automaton. The generated automaton 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 3.2.

generates L only once—state by state—checking every time, whether the new state l is not already present in a set of states Q_L of L.

Due to the nature of lasso automata, the successive product states generate the lasso automata very similar to L. We just need to append new states to Q_L. 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 $l \notin Q_L$, we add l to Q_L and continue with the following states l' until we either create an entirely new loop in L or generate $l' \in Q_L$. If $l \in Q_L$, we can stop generating l' for q as $\forall l'(l' \in Q_L)$.

3.3 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 it and resolve emptiness test for each *branch* of the potential product automaton.

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  sat ← False
5  solved ← ∅
6  while W ≠ ∅ do
7      picklast [q1, q2] from W
8      add [q1, q2] to solved
9      sat ← satisfiable([q1, q2])
10     if sat then
11         add [q1, q2] to Q
12         if q1 ∈ F1 and q2 ∈ F2 then
13             add [q1, q2] to F
14         forall a ∈ Σ do
15             forall q1' ∈ δ1(q1, a), q2' ∈ δ2(q2, a)
                   do
16                 if [q1', q2'] ∉ solved and
                       [q1', q2'] ∉ W then
17                     add [q1', q2'] to W
18                 add [q1', q2'] to δ([q1, q2], a)
```

Algorithm 2: Product construction with length abstraction

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 q is picked from W. Instead of immediately generating new successive product states, we test q for satisfiability of length constraints of recognized words from q. On line 9, we check for satisfiability of formulae generated from q. If the process shows formulae are satisfiable, i.e., there will be an accepting run using q (see line 10), we add q to Q, possibly to F and generate its successive product states q'.

The formulae are generated using lasso automata for both original automata. For every state, we get one or more formulae in the form $\phi : \exists k(|w| = a + b \cdot k)$, where $|w|$ is a length of recognized word, a is the length of a handle to a certain accepting state, and b is the length of a loop to return to this particular accepting state going through the loop. k is the amount 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 the lasso automaton), we append these formulae with *logical or* ($\vee$), then compare these with the formulae from the other lasso automaton for the other initial finite automaton using SMT solver.

To better demonstrate our solution, the second automaton we will be working with is finite automaton A_2 from Figure 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 4.

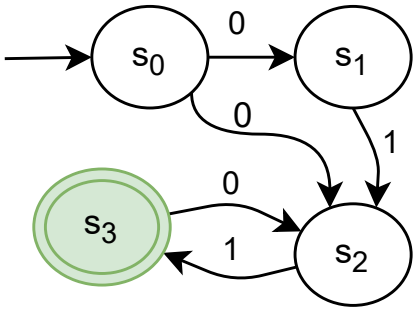


Figure 4. Non-deterministic finite automaton A_2

[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 7)—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 explained in Section 3.4.

In Figure 5, there is its lasso automaton A'_2, which we will be using together with the lasso automaton shown earlier in Figure 3 for computation of recognized word lengths.

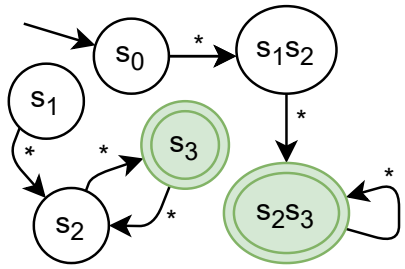


Figure 5. Lasso automaton A'_2 for the original NFA A_2

For automaton A_1 for the initial state (we start computing lengths as if the state q_0 is the initial state) from A'_1, we get formula ϕ[6]. For automaton A_2 for the initial state from the lasso automaton A'_2, we get formula ψ[7].

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

When we compare ϕ and ψ, we get:

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

We try to find values of k and l such that some of the expressions on the left and on the right side of the equation are equal. This equation is handed 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 of these automata, when there is even no word fulfilling the length requirements. In this case, we have successfully reduced the generated state space by omitting this particular tested product state and any further product states, which would be later normally generated from these product states and its successors (assuming the transition symbols correspond).

In Figure 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

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

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

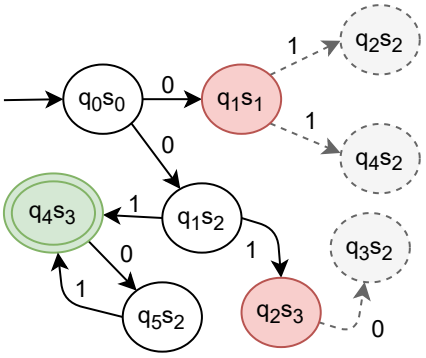


Figure 6. Constructed product automaton with depiction of our optimization.

algorithm omits any successive product states—dashed states (such as q_4s_2 or q_3s_2), which are generated in the basic 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.

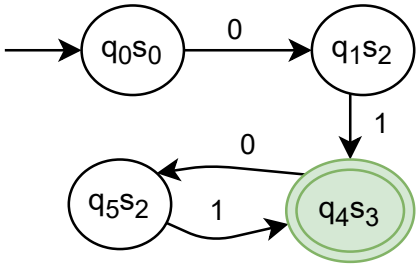


Figure 7. Final product generated by our product construction algorithm optimizing its process by omitting unnecessary states on the way.

As you can notice in Figure 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 the result of formulae being unsatisfiable, we do not need to construct any following states and check their products for satisfiability and therefore, we are able to determine whether such *branches* of automata have an empty intersections 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 (the result of the length abstraction satisfiability check is *sat*), 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 the length abstraction satisfiability check returns *unsat*.

3.4 Further Optimization with Skipping Satisfiable States

When we take a new product state q from work set W and check for satisfiability with formulae for q being satisfiable, it is time to add to W all of the possible successive product states q'. When q generates only a single q', we can say with certainty formulae for q' are satisfiable too as there is only a single branch in the automaton leading from q to an accepting state (through q'). Product state q' is *skippable*, if exists a satisfiable q whose only successor is q'. We add q' to W with the information of being skippable. If q' is already in W, we append the information to q' in W.

We will skip checking for the satisfiability when we pick q' from W and immediately check for final states and generate the successive product states. This optimization will save us generating the formulae for q' and testing the formulae in the SMT solver for their satisfiability and even possibly reducing the amount of states generated for both our lasso automata.

If our original automata have long lines (with non-splitting branches), this will prove extremely useful, because only a few proper iterations with formulae computing and executing SMT solver will be executed. In Algorithm 3 is depicted the application of skipping satisfiable states. The line 9 from Algorithm 2 is substituted with the contents of Algorithm 3.

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.

```
if not skippable([q1, q2]) then
    sat ← satisfiable([q1, q2])
else
    sat ← True
```

Algorithm 3: Substitution of line 9 in Algorithm 2 with skipping satisfiable states

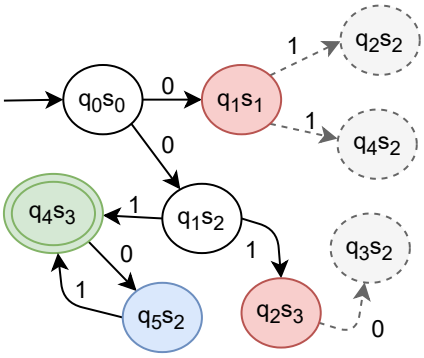


Figure 8. Constructed product automaton with depiction of skipping states optimization

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 8 is such 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_4s3 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 omit the entire branch (if *unsat* is returned) or skip checking satisfiability in the entire branch (if *sat* is returned).

4. Experiments and Results

The reference implementation[8] of optimization, 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 GitHub repository[9]. 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 amount 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.

The following graphs shows the results for both the emptiness test and full product construction. The graph in Figure 9 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 10 shows the same data, only for 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 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 amounts of transitions from every state causing large amounts 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 amount 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 generation is complete. Therefore, the lasso automata do not affect how efficiently we work with the generated product. Nevertheless, the amount of generated lasso states in the process of deciding inter-

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

[9] https://github.com/Adda0/optimizing_automata_product_construction_and_emptiness_test

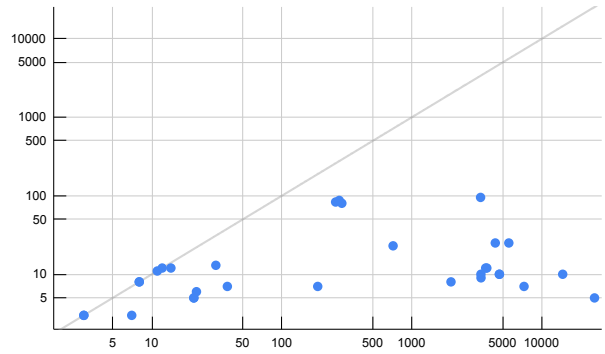


Figure 9. Emptiness test

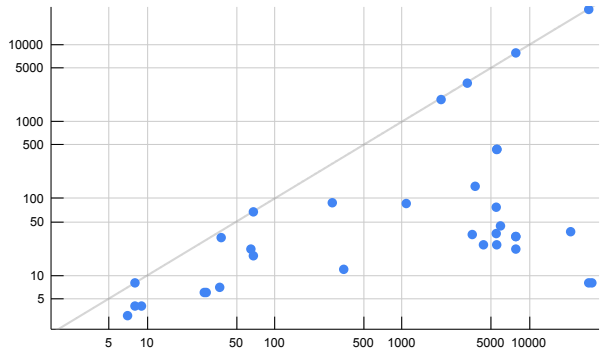


Figure 10. Full product construction

Figure 11. 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).

section emptiness test matters. For different automata, the generated state space varies. For state space sizes of lasso automata in our experiments see the GitHub repository[10].

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 computing the emptiness test of these states using length abstraction over the original automata using lasso automata.

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 most usually used operations on automata intersection. And because length abstraction considers over-approximation of possible products, our algorithm is safe to use for any uses resolving automata intersection.

We have not encountered similar approaches to product construction optimization using length or other abstraction to compare our results with. It might be worth investing into combining our orthogonal approach with other existing algorithm to see how the generated product state space is affected. We are talking about abstraction techniques such as CEGAR [8] and predicate abstraction [9, 10], IMPACT [11], possibly IC3/PDR [12, 13]. All of 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, 15, 16] and on the interpolation-based approach of McMillan [17, 18].

We can also compute Parikh images using Parikh theorem [19] instead of just lengths of recognized words. Parikh images give us additional information about every product state. This allows more precise estimation of possible accepted words by both automata and deciding the emptiness test of their intersection more effectively. Computation of Parikh image is time consuming, but might further minimize the product construction state space.

[10] https://github.com/Adda0/optimizing_automata_product_construction_and_emptiness_test/tree/master/results

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] Tomás Fiedor, Lukás Holík, Ondrej Lengál, and Tomás Vojnar. Nested antichains for WS1S. *Acta Informatica*, 56(3):205–228, 2019.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2014.

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

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

[8] 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.

[9] 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.

[10] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

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

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

[13] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, pages 173–180. IEEE Computer Society, 2007.

[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] Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.

[16] Arlen Cox and Jason Leasure. Model checking regular language constraints. *CoRR*, abs/1708.09073, 2017.

[17] Nina Amla and Kenneth L. McMillan. Combining abstraction refinement and sat-based model checking. In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 405–419. Springer, 2007.

[18] 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.

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