When Simulation Meets Antichains
(on Checking Language Inclusion of NFAs)

Parosh Aziz Abdulla[1], Yu-Fang Chen[1], Lukáš Holík[2], Richard Mayr[3], and Tomáš Vojnar[2]

[1]Uppsala University [2]Brno University of Technology [3]University of Edinburgh

Abstract. We describe a new and more efficient algorithm for checking universality and language inclusion on nondeterministic finite word automata (NFA) and tree automata (TA). To the best of our knowledge, the antichain-based approach proposed by Wulf et al. was the most efficient one so far. Our idea is to exploit a simulation relation on the states of finite automata to accelerate the antichain-based algorithms. Normally, a simulation relation can be obtained fairly efficiently, and it can help the antichain-based approach to prune out a large portion of unnecessary search paths. We evaluate the performance of our new method on NFA/TA obtained from random regular expressions and from the intermediate steps of regular model checking. The results show that our approach significantly outperforms the previous antichain-based approach in most of the experiments.

1 Introduction

The language inclusion problem for regular languages is important in many application domains, e.g., formal verification. Many verification problems can be formulated as a language inclusion problem. For example, one may describe the actual behaviors of an implementation in an automaton $\mathcal{A}$ and all of the behaviors permitted by the specification in another automaton $\mathcal{B}$. Then, the problem of whether the implementation meets the specification is equivalent to the problem $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

Methods for proving language inclusion can be categorized into two types: those based on *simulation* (e.g., [6]) and those based on the *subset construction* (e.g., [5, 8–10]). Simulation-based approaches first compute a simulation relation on the states of two automata $\mathcal{A}$ and $\mathcal{B}$ and then check if all initial states of $\mathcal{A}$ can be simulated by some initial state of $\mathcal{B}$. Since simulation can be computed in polynomial time, simulation-based methods are usually very efficient. Their main drawback is that they are incomplete. Simulation preorder implies language inclusion, but not vice-versa.

On the other hand, methods based on the subset construction are complete but inefficient because in many cases they will cause an exponential blow up in the number of states. Recently, Wulf et al. [11] proposed the *antichain-based* approach. To the best of our knowledge, it was the most efficient one among all of the methods based on the subset construction. Although the antichain-based method significantly outperforms the classical subset construction, in many cases, it still sometimes suffers from the exponential blow up problem.

In this paper, we describe a new approach that nicely combines the simulation-based and the antichain-based approaches. The computed simulation relation is used for pruning out unnecessary search paths of the antichain-based method.

To simplify the presentation, we first consider the problem of checking universality for a word automaton $\mathcal{A}$. In a similar manner to the classical subset construction, we

start from the set of initial states and search for sets of states (here referred to as *macro-states*) which are not accepting (i.e., we search for a counterexample of universality). The key idea is to define an "easy-to-check" ordering $\preceq$ on the states of $\mathcal{A}$ which implies language inclusion (i.e., $p \preceq q$ implies that the language of the state p is included in the language of the state q). From $\preceq$, we derive an ordering on macro-states which we use in two ways to optimize the subset construction: (1) searching from a macro-state needs not continue in case a smaller macro-state has already been analyzed; and (2) a given macro-state is represented by (the subset of) its maximal elements. In this paper, we take the ordering $\preceq$ to be the well-known maximal simulation relation on the automaton $\mathcal{A}$. In fact, the anti-chain algorithm of [11] coincides with the special case where the ordering $\preceq$ is the identity relation.

Subsequently, we describe how to generalize the above approach to the case of checking language inclusion between two automata $\mathcal{A}$ and $\mathcal{B}$, by extending the ordering to pairs each consisting of a state of $\mathcal{A}$ and a macro-state of $\mathcal{B}$.

In the second part of the paper, we extend our algorithms to the case of tree automata. First, we define the notion of *open trees* which we use to characterize the languages defined by tuples of states of the tree automaton. We identify here a new application of the so called upward simulation relation from [1]. We show that it implies (open tree) language inclusion, and we describe how we can use it to optimize existing algorithms for checking the universality and language inclusion properties.

We have implemented our algorithms and carried out an extensive experimentation using NFA obtained from several different sources. These include NFA from random regular expressions and also 1069 pairs of NFA generated from the intermediate steps of abstract regular model checking [4] while verifying the correctness of the bakery algorithm, a producer-consumer system, the bubble sort algorithm, an algorithm that reverses a circular list, and a Petri net model of the readers/writers protocol. We have also considered tree-automata derived from intermediate steps of abstract regular tree model checking. The experiments show that our approach significantly outperforms the previous antichain-based approach in almost all of the considered cases. (Furthermore, in those cases where simulation is sufficient to prove language inclusion, our algorithm has polynomial running time.)

The remainder of the paper is organized as follows. Section 2 contains some basic definitions. In Section 3, we begin the discussion by applying our idea to solve the universality problem for NFA. The problem is simpler than the language inclusion problem and thus we believe that presenting our universality checking algorithm first makes it easier for the reader to grasp the idea. The correctness proof of our universality checking algorithm is given in Section 4. In Section 5 we discuss our language inclusion checking algorithm for NFA. Section 6 defines basic notations for tree automata and in Section 7, we present the algorithms for checking universality and language inclusion for tree automata. The experimental results are described in Section 8. Finally, in Section 9, we conclude the paper and discuss further research directions.

2 Preliminaries

A *Nondeterministic Finite Automaton (NFA)* $\mathcal{A}$ is a tuple $(\Sigma, Q, I, F, \delta)$ where: Σ is an alphabet, Q is a finite set of states, $I \subseteq Q$ is a non-empty set of *initial* states, $F \subseteq Q$ is a

set of *final* states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. For convenience, we use $p \xrightarrow{a} q$ to denote the transition from the state p to the state q with the label a.

A word $u = u_1 \dots u_n$ is accepted by $\mathcal{A}$ from the state q_0 if there exists a sequence $q_0 u_1 q_1 u_2 \dots u_n q_n$ such that $q_n \in F$ and $q_{j-1} \xrightarrow{u_j} q_j$ for all $0 < j \leq n$. Define $\mathcal{L}(\mathcal{A})(q) := \{u \mid u \text{ is accepted by } \mathcal{A} \text{ from the state } q\}$ (the language of the state q in $\mathcal{A}$). Define the language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ as $\bigcup_{q \in I} \mathcal{L}(\mathcal{A})(q)$. We say that $\mathcal{A}$ is *universal* if $\mathcal{L}(\mathcal{A}) = \Sigma^*$. Let $\mathcal{A} = (\Sigma, Q_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \delta_\mathcal{A})$ and $\mathcal{B} = (\Sigma, Q_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \delta_\mathcal{B})$ be two NFAs. Define their union automaton $\mathcal{A} \cup \mathcal{B} := (\Sigma, Q_\mathcal{A} \cup Q_\mathcal{B}, I_\mathcal{A} \cup I_\mathcal{B}, F_\mathcal{A} \cup F_\mathcal{B}, \delta_\mathcal{A} \cup \delta_\mathcal{B})$. We define the post-image of a state $Post(p) := \{p' \mid \exists a \in \Sigma : (p, a, p') \in \delta\}$.

A *simulation* on $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is a relation $\preceq \subseteq Q \times Q$ such that $p \preceq r$ only if (i) $p \in F \implies r \in F$ and (ii) for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$. It can be shown that for each automaton $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, there exists a unique maximal simulation. The following is a well-known lemma.

Lemma 1. *Given a simulation $\preceq$ on an NFA $\mathcal{A}$, $p \preceq r \implies \mathcal{L}(\mathcal{A})(p) \subseteq \mathcal{L}(\mathcal{A})(r)$.*

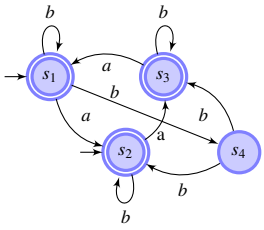
For convenience, we call a set of states in $\mathcal{A}$ a *macro-state*, i.e., a macro-state is a subset of Q. A macro-state is *accepting* if it contains at least one accepting state, otherwise it is *rejecting*. For a macro-state P, define $\mathcal{L}(\mathcal{A})(P) := \bigcup_{p \in P} \mathcal{L}(\mathcal{A})(p)$. We say that a macro-state P is universal if $\mathcal{L}(\mathcal{A})(P) = \Sigma^*$. For two macro-states P and R, we write $P \preceq^{\forall\exists} R$ as a shorthand for $\forall p \in P . \exists r \in R : p \preceq r$. We define the post-image of a macro-state $Post(P) := \{P' \mid \exists a \in \Sigma : P' = \{p' \mid \exists p \in P : (p, a, p') \in \delta\}\}$. We use $\mathcal{A}^\subseteq$ to denote the set of relations over the states of $\mathcal{A}$ that implies language inclusion, i.e., if $\preceq \in \mathcal{A}^\subseteq$, then we have $p \preceq r \implies \mathcal{L}(\mathcal{A})(p) \subseteq \mathcal{L}(\mathcal{A})(r)$.

3 Universality of NFAs

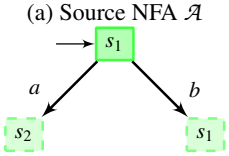
The *universality problem* for an NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is to decide whether $\mathcal{L}(\mathcal{A}) = \Sigma^*$. The problem is PSPACE-complete. The classical algorithm for the problem first determinizes $\mathcal{A}$ with the subset construction and then checks if every reachable macro-state is accepting. The algorithm is inefficient since in many cases the determinization will cause an exponential blow-up in the number of states. Note that for universality checking, we can stop the subset construction immediately and conclude that $\mathcal{A}$ is not universal whenever a rejecting macro-state is encountered. An example of a run of this algorithm is given in Fig. 1. The automaton $\mathcal{A}$ used in Fig. 1 is universal because all reachable macro-states are accepting.

In this section, we propose a more efficient approach to universality checking. In a similar manner to the classical algorithm, we run the subset construction procedure and check if any rejecting macro-state is reachable. However, our algorithm augments the subset construction with two optimizations, henceforth referred to as *Optimization 1* and *Optimization 2*, respectively.

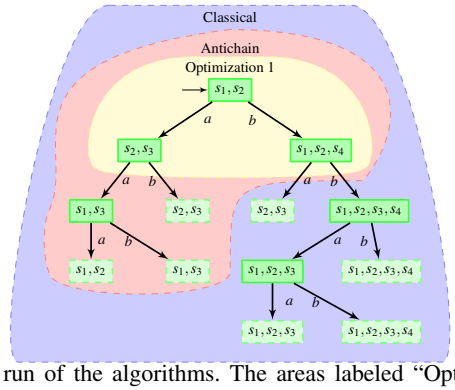
Optimization 1 is based on the fact that if the algorithm encounters a macro-state R whose language is a superset of the language of a visited macro-state P, then there is no need to continue the search from R. The intuition behind this is that if a word is not accepted from R, then it is also not accepted from P. For instance, in Fig. 1(b), the search needs not continue from the macro-state $\{s_2, s_3\}$ since its language is a superset of the language of the initial macro-state $\{s_1, s_2\}$. However, in general it is difficult to



(a) Source NFA $\mathcal{A}$



(c) Optimization 1 and 2



(b) A run of the algorithms. The areas labeled "Optimization 1", "Antichain", "Classical" are the macro-states generated by our approach with the maximal simulation and Optimization 1, the antichain-based approach, and the classical approach, respectively.

Fig. 1. Universality Checking Algorithms

check if $\mathcal{L}(\mathcal{A})(P) \subseteq \mathcal{L}(\mathcal{A})(R)$ before the resulting DFA is completely built. Therefore, we suggest to use an easy-to-compute alternative based on the following lemma.

Lemma 2. *Let P, R be two macro-states, $\mathcal{A}$ be an NFA, and $\preceq$ be a relation in $\mathcal{A}^{\subseteq}$. Then, $P \preceq^{\forall\exists} R$ implies $\mathcal{L}(\mathcal{A})(P) \subseteq \mathcal{L}(\mathcal{A})(R)$.*

Note that in Lemma 2, $\preceq$ can be any relation on the states of $\mathcal{A}$ that implies language inclusion. This includes any simulation relation (Lemma 1). When $\preceq$ is the maximal simulation or the identity relation, it can be efficiently obtained from $\mathcal{A}$ before the subset construction algorithm is triggered and used to prune out unnecessary search paths.

An example of how the described optimization can help is given in Fig. 1(b). If $\preceq$ is the identity, the universality checking algorithm will not continue the search from the macro-state $\{s_1, s_2, s_4\}$ because it is a superset of the initial macro-state. In fact, the antichain-based approach [11] can be viewed as a special case of our approach when $\preceq$ is the identity. Notice that, in this case, only 7 macro-states are generated (the classical algorithm generates 13 macro-states). When $\preceq$ is the maximal simulation, we do not need to continue from the macro-state $\{s_2, s_3\}$ either because $s_1 \preceq s_3$ and hence $\{s_1, s_2\} \preceq^{\forall\exists} \{s_2, s_3\}$. In this case, only 3 macro-states are generated. As we can see from the example, a better reduction of the number of generated states can be achieved when a weaker relation (e.g., the maximal simulation) is used.

Optimization 2 is based on the observation that $\mathcal{L}(\mathcal{A})(P) = \mathcal{L}(\mathcal{A})(P \setminus \{p_1\})$ if there is some $p_2 \in P$ with $p_1 \preceq p_2$. This fact is a simple consequence of Lemma 2 (note that $P \preceq^{\forall\exists} P \setminus \{p_1\}$). Since the two macro-states P and $P \setminus \{p_1\}$ have the same language, if a word is not accepted from P, it is not accepted from $P \setminus \{p_1\}$ either. On the other hand, if all words in Σ^* can be accepted from P, then they can also be accepted from $P \setminus \{p_1\}$. Therefore, it is safe to replace the macro-state P with $P \setminus \{p_1\}$.

Consider the example in Fig. 1. If $\preceq$ is the maximal simulation relation, we can remove the state s_2 from the initial macro-state $\{s_1, s_2\}$ without changing its language,

Algorithm 1: *Universality Checking*

Input: An NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ and a relation $\preceq \in \mathcal{A}^{\subseteq}$.
Output: TRUE if $\mathcal{A}$ is universal. Otherwise, FALSE.

```
1  if I is rejecting then return FALSE;
2  Processed:=∅;
3  Next:={Minimize(I)};
4  while Next ≠ ∅ do
5      Pick and remove a macro-state R from Next and move it to Processed;
6      foreach P ∈ {Minimize(R') | R' ∈ Post(R)} do
7          if P is rejecting then return FALSE;
8          else if ∄S ∈ Processed ∪ Next s.t. S ⪯^{∀∃} P then
9              Remove all S from Processed ∪ Next s.t. P ⪯^{∀∃} S;
10             Add P to Next;
11 return TRUE
```

because $s_2 \preceq s_1$. This change will propagate to all the searching paths. With this optimization, our approach will only generates 3 macro-states, all of which are singletons. The result after apply the two optimizations are applied is shown in Fig. 1(c).

Algorithm 1 describes our approach in pseudocode. In this algorithm, the function $\mathit{Minimize}(R)$ implements Optimization 2. The function does the following: it chooses a new state r_1 from R, removes r_1 from R if there exists a state r_2 in R such that $r_1 \preceq r_2$, and then repeats the procedure until all of the states in R are processed. Lines 8–10 of the algorithm implement Optimization 1. Overall, the algorithm works as follows. Till the set *Next* of macro-states waiting to be processed is non-empty (or a rejecting macro-state is found), the algorithm chooses one macro-state from *Next*, and moves it to the *Processed* set. Moreover, it generates all successors of the chosen macro-state, minimizes them, and adds them to *Next* unless there is already some $\preceq^{\forall\exists}$-smaller macro-state in *Next* or in *Processed*. If a new macro-state is added to *Next*, the algorithm at the same time removes all $\preceq^{\forall\exists}$-bigger macro-states from both *Next* and *Processed*. Note that the pruning of the *Next* and *Processed* sets together with checking whether a new macro-state should be added into *Next* can be done within a single iteration through *Next* and *Processed*. We discuss correctness of the algorithm in the next section.

4 Correctness of the Optimized Universality Checking

In this section, we prove correctness of Algorithm 1. Due to the space limitation, we only present an overview. A more detailed proof can be found in Appendix A. Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be the input automaton. We first introduce some definitions and notations that will be used in the proof. For a macro-state P, define $\mathit{Dist}(P) \in \mathbb{N} \cup \{\infty\}$ as the length of the shortest word in Σ^* that is not in $\mathcal{L}(\mathcal{A})(P)$ (if $\mathcal{L}(\mathcal{A})(P) = \Sigma^*$, $\mathit{Dist}(P) = \infty$). For a set of macro-states *MStates*, the function $\mathit{Dist}(\mathit{MStates}) \in \mathbb{N} \cup \{\infty\}$ returns the length of the shortest word in Σ^* that is not in the language of some macro-state in *MStates*. More precisely, if $\mathit{MStates} = \emptyset$, $\mathit{Dist}(\mathit{MStates}) = \infty$, otherwise, $\mathit{Dist}(\mathit{MStates}) = \mathit{min}_{P \in \mathit{MStates}} \mathit{Dist}(P)$. The predicate $\mathit{Univ}(\mathit{MStates})$ is true if and only if all the macro-states in *MStates* are universal, i.e., $\forall P \in \mathit{MStates} : \mathcal{L}(\mathcal{A})(P) = \Sigma^*$.

Lemma 3 describes the invariants used to prove the partial correctness of Alg. 1.

Lemma 3. *The below two loop invariants hold in Algorithm 1:*

1. $\neg Univ(Processed \cup Next) \implies \neg Univ(\{I\})$.
2. $\neg Univ(\{I\}) \implies Dist(Processed) > Dist(Next)$.

Due to the finite number of macro-states, we can show that Algorithm 1 eventually terminates. Algorithm 1 returns FALSE only if either the set of initial states is rejecting, or the minimized version of some successor R' of a macro-state R chosen from *Next* on line 5 is found rejecting. In the latter case, due to Lemma 2, R' is also rejecting. Then, R is non-universal, and hence $Univ(Processed \cup Next)$ is false. By Lemma 3 (Invariant 1), we have $\mathcal{A}$ is not universal. The algorithm returns TRUE only when *Next* becomes empty. When *Next* is empty, $Dist(Processed) > Dist(Next)$ is not true. Therefore, by Lemma 3 (Invariant 2), $\mathcal{A}$ is universal. This gives the following theorem.

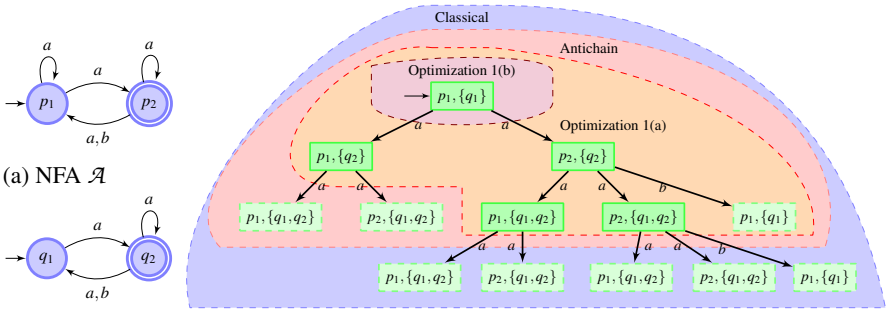
Theorem 1. *Algorithm 1 always terminates and returns* TRUE *iff the input automaton $\mathcal{A}$ is universal.*

5 The Language Inclusion Problem

The technique described in Section 3 can be generalized to solve the *language-inclusion problem*. Let $\mathcal{A}$ and $\mathcal{B}$ be two NFAs. The *language inclusion problem* for $\mathcal{A}$ and $\mathcal{B}$ is to decide whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. This problem is also PSPACE-complete. The classical algorithm for solving this problem builds on-the-fly the product automaton $\mathcal{A} \times \overline{\mathcal{B}}$ of $\mathcal{A}$ and the complement of $\mathcal{B}$ and searches for an accepting state. A state in the product automaton $\mathcal{A} \times \overline{\mathcal{B}}$ is a pair (p,P) where p is a state in $\mathcal{A}$ and P is a macro-state in $\mathcal{B}$. For convenience, we call such a pair (p,P) a *product-state*. A product-state is accepting iff p is an accepting state in $\mathcal{A}$ and P is a rejecting macro-state in $\mathcal{B}$. We use $\mathcal{L}(\mathcal{A},\mathcal{B})(p,P)$ to denote the language of the product-state (p,P) in $\mathcal{A} \times \overline{\mathcal{B}}$. The language of $\mathcal{A}$ is not contained in the language of $\mathcal{B}$ iff there exists some accepting product-state (p,P) reachable from some initial product-state. Indeed, $\mathcal{L}(\mathcal{A},\mathcal{B})(p,P) = \mathcal{L}(\mathcal{A})(p) \setminus \mathcal{L}(\mathcal{B})(P)$, and the language of $\mathcal{A} \times \overline{\mathcal{B}}$ consists of words which can be used as witnesses of the fact that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ does not hold. In a similar manner to universality checking, the algorithm can stop the search immediately and conclude that the language inclusion does not hold whenever an accepting product-state is encountered. An example of a run of the classical algorithm is given in Fig. 2. We find that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ is true and the algorithm generates 13 product-states (Fig. 2(c), the area labeled "Classical").

Optimization 1 that we use for universality checking can be generalized for language inclusion checking as follows. Let $\mathcal{A} = (\Sigma, Q_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \delta_\mathcal{A})$ and $\mathcal{B} = (\Sigma, Q_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \delta_\mathcal{B})$ be two NFAs such that $Q_\mathcal{A} \cap Q_\mathcal{B} = \emptyset$. We denote by $\mathcal{A} \cup \mathcal{B}$ the NFA $(\Sigma, Q_\mathcal{A} \cup Q_\mathcal{B}, I_\mathcal{A} \cup I_\mathcal{B}, F_\mathcal{A} \cup F_\mathcal{B}, \delta_\mathcal{A} \cup \delta_\mathcal{B})$. Let $\preceq$ be a relation in $(\mathcal{A} \cup \mathcal{B})^{\subseteq}$. During the process of constructing the product automaton and searching for an accepting product-state, we can stop the search from a product-state (p,P) if (a) there exists some visited product-state (r,R) such that $p \preceq r$ and $R \preceq^{\forall\exists} P$, or (b) $\exists p' \in P : p \preceq p'$. Optimization 1(a) is justified by Lemma 4, which is very similar to Lemma 2 for universality checking.

Lemma 4. *Let $\mathcal{A}$, $\mathcal{B}$ be two NFAs, (p,P), (r,R) be two product-states, where p, r are states in $\mathcal{A}$ and P, R are macro-states in $\mathcal{B}$, and $\preceq$ be a relation in $(\mathcal{A} \cup \mathcal{B})^{\subseteq}$. Then, $p \preceq r$ and $R \preceq^{\forall\exists} P$ implies $\mathcal{L}(\mathcal{A},\mathcal{B})(p,P) \subseteq \mathcal{L}(\mathcal{A},\mathcal{B})(r,R)$.*



(a) NFA $\mathcal{A}$

(b) NFA $\mathcal{B}$

(c) A run of the algorithms while checking $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

Fig. 2. Language Inclusion Checking Algorithms

By the above lemma, if a word takes the product-state (p,P) to an accepting product-state, it will also take (r,R) to an accepting product-state. Therefore, we do not need to continue the search from (p,P).

Let us use Fig. 2(c) to illustrate Optimization 1(a). As we mentioned, the antichain-based approach can be viewed as a special case of our approach when $\preceq$ is the identity. When $\preceq$ is the identity, we do not need to continue the search from the product-state $(p_2,\{q_1,q_2\})$ because $\{q_2\} \subseteq \{q_1,q_2\}$. In this case, the algorithm generates 8 product-states (Fig. 2(c), the area labeled "Antichain"). In the case that $\preceq$ is the maximal simulation, we do not need to continue the search from product-states $(p_1,\{q_2\})$, $(p_1,\{q_1,q_2\})$, and $(p_2,\{q_1,q_2\})$ because $q_1 \preceq q_2$ and the algorithm already visited the product-states $(p_1,\{q_1\})$ and $(p_2,\{q_2\})$. Hence, the algorithm generates only 6 product-states (Fig. 2(c), the area labeled "Optimization 1(a)").

If the condition of Optimization 1(b) holds, we have that the language of p (w.r.t. $\mathcal{A}$) is a subset of the language of P (w.r.t. $\mathcal{B}$). In this case, for any word that takes p to an accepting state in $\mathcal{A}$, it also takes P to an accepting macro-state in $\mathcal{B}$. Hence, we do not need to continue the search from the product-state (p,P) because all of its successor states are rejecting product-states. Consider again the example in Fig. 2(c). With Optimization 1(b), if $\preceq$ is the maximal simulation on the states of $\mathcal{A} \cup \mathcal{B}$, we do not need to continue the search from the first product-state $(p_1,\{q_1\})$ because $p_1 \preceq q_1$. In this case, the algorithm can conclude that the language inclusion holds immediately after the first product-state is generated (Fig. 2(c), the area labeled "Optimization 1(b)").

Observe that from Lemma 4, it holds that for any product-state (p,P) such that $p_1 \preceq p_2$ for some $p_1,p_2 \in P$, $\mathcal{L}(\mathcal{A},\mathcal{B})(p,P) = \mathcal{L}(\mathcal{A},\mathcal{B})(p,P \setminus \{p_1\})$ (as $P \preceq^{\forall\exists} P \setminus \{p_1\}$). Optimization 2 that we used for universality checking can therefore be generalized for language inclusion checking too.

We give the pseudocode of our optimized inclusion checking in Algorithm 2, which is a straightforward extension of Algorithm 1. In the algorithm, the definition of the $\mathit{Minimize}(R)$ function is the same as what we have defined in Algorithm 1. The function $\mathit{Initialize}(\mathit{PStates})$ applies Optimization 1 on the set of product-states $\mathit{PStates}$ to avoid unnecessary searching. More precisely, it returns a maximal subset of $\mathit{PStates}$ such that (1) for any two elements (p,P), (q,Q) in the subset, $p \not\preceq q \vee Q \not\preceq^{\forall\exists} P$ and (2) for any element (p,P) in the subset, $\forall p' \in P : \ p \not\preceq p'$. We define the post-image of a product-state $\mathit{Post}((p,P)) := \{(p',P') \mid \exists a \in \Sigma : (p,a,p') \in \delta, P' = \{p'' \mid \exists p \in P : (p,a,p'') \in \delta\}\}$.

Algorithm 2: *Language Inclusion Checking*

Input: NFA $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}}, \delta_{\mathcal{A}})$, $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$. A relation $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.

if *there is an accepting product-state in* $\{(i, I_{\mathcal{B}}) \mid i \in I_{\mathcal{A}}\}$ **then return** *FALSE*;
Processed:=$\emptyset$;
Next:= *Initialize*$(\{(i, Minimize(I_{\mathcal{B}})) \mid i \in I_{\mathcal{A}}\})$;
while *Next* $\neq \emptyset$ **do**
    Pick and remove a product-state (r, R) from *Next* and move it to *Processed*;
    foreach $(p, P) \in \{(r', Minimize(R')) \mid (r', R') \in Post((r, R))\}$ **do**
        if (p, P) *is an accepting product-state* **then return** *FALSE*;
        else if $\nexists p' \in P$ *s.t.* $p \preceq p'$ **then**
            if $\nexists (s, S) \in Processed \cup Next$ *s.t.* $p \preceq s \wedge S \preceq^{\forall\exists} P$ **then**
                Remove all (s, S) from $Processed \cup Next$ s.t. $s \preceq p \wedge P \preceq^{\forall\exists} S$;
                Add (p, P) to *Next*;
return *TRUE*

Correctness: Define $Dist(P) \in \mathbb{N} \cup \{\infty\}$ as the length of the shortest word in the language of the product-state P or ∞ if the language of P is empty. The value $Dist(PStates) \in \mathbb{N} \cup \{\infty\}$ is the length of the shortest word in the language of some product-state in *PStates* or ∞ if *PStates* is empty. The predicate $Incl(PStates)$ is true iff for all product-states (p, P) in *PStates*, $\mathcal{L}(\mathcal{A})(p) \subseteq \mathcal{L}(\mathcal{B})(P)$. The correctness of Algorithm 2 can now be proved in a very similar way to Algorithm 1, using the below invariants:

1. $\neg Incl(Processed \cup Next) \implies \neg Incl(\{(i, I_{\mathcal{B}}) \mid i \in I_{\mathcal{A}}\})$.
2. $\neg Incl(\{(i, I_{\mathcal{B}}) \mid i \in I_{\mathcal{A}}\}) \implies Dist(Processed) > Dist(Next)$.

6 Tree Automata Preliminaries

To be able to present a generalization of the above methods for the domain of tree automata, we now introduce some needed preliminaries on tree automata.

A *ranked alphabet* Σ is a set of symbols together with a ranking function $\# : \Sigma \to \mathbb{N}$. For $a \in \Sigma$, the value $\#(a)$ is called the *rank* of a. For any $n \geq 0$, we denote by Σ_n the set of all symbols of rank n from Σ. Let ε denote the empty sequence. A *tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions: (1) $dom(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$ and (2) for each $v \in dom(t)$, if $\#(t(v)) = n \geq 0$, then $\{i \mid vi \in dom(t)\} = \{1, \ldots, n\}$. Each sequence $v \in dom(t)$ is called a *node* of t. For a node v, we define the i^{th} *child* of v to be the node vi, and the i^{th} *subtree* of v to be the tree t' such that $t'(v') = t(viv')$ for all $v' \in \mathbb{N}^*$. A *leaf* of t is a node v which does not have any children, i.e., there is no $i \in \mathbb{N}$ with $vi \in dom(t)$. We denote by $T(\Sigma)$ the set of all trees over the alphabet Σ.

A (finite, non-deterministic, bottom-up) *tree automaton* (abbreviated as TA in the sequel) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1, \ldots, q_n), a, q)$ where $q_1, \ldots, q_n, q \in Q$, $a \in \Sigma$, and $\#(a) = n$. We use $(q_1, \ldots, q_n) \xrightarrow{a} q$ to denote that $((q_1, \ldots, q_n), a, q) \in \Delta$. In the special case where $n = 0$, we speak about the so-called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{a} q$.

Let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a TA. A *run* of $\mathcal{A}$ over a tree $t \in T(\Sigma)$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $v \in dom(t)$ of arity $\#(t(v)) = n$ where $q = \pi(v)$, if $q_i = \pi(vi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1, \ldots, q_n) \xrightarrow{t(v)} q$. We write $t \xRightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \xRightarrow{\pi} q$ for some run π. The *language* accepted by a state q is defined by $\mathcal{L}(\mathcal{A})(q) = \{t \mid t \Longrightarrow q\}$, while the *language* of $\mathcal{A}$ is defined by $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} \mathcal{L}(\mathcal{A})(q)$.

7 Universality and Language Inclusion of Tree Automata

To optimize universality and inclusion checking on word automata, we used relations that imply language inclusion. For the case of universality and inclusion checking on tree automata, we now propose to use relations that imply inclusion of languages of the so called "open" trees (i.e., "leafless" trees or equivalently trees whose leaves are replaced by a special symbol denoting a "hole") that are accepted from tuples of tree automata states. We formally define the notion below. Notice that in contrast to the notion of a language accepted from a state of a word automaton, which refers to possible "futures" of the state, the notion of a language accepted at a state of a TA refers to possible "pasts" of the state. Our notion of languages of open trees accepted from tuples of tree automata states speaks again about the future of states, which turns out useful when trying to optimize the (antichain-based) subset construction for TA.

Consider a special symbol $\Box \notin \Sigma$ with rank 0, called a *hole*. An *open tree* over Σ is a tree over $\Sigma \cup \Box$ such that all its leaves are labeled[1] by $\Box$. We use $T^{\Box}(\Sigma)$ to denote the set of all open trees over Σ. Given states $q_1, \ldots, q_n \in Q$ and an open tree t with leaves $v_1, \ldots, v_n$, a run π of $\mathcal{A}$ on t from $(q_1, \ldots, q_n)$ is defined in a similar way as the run on a tree except that for each leaf v_i, $1 \leq i \leq n$, we have $\pi(v_i) = q_i$. We use $t(q_1, \ldots, q_n) \xRightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ on t from $(q_1, \ldots, q_n)$ such that $\pi(\varepsilon) = q$. The notation $t(q_1, \ldots, q_n) \Longrightarrow q$ is explained in a similar manner to runs on trees. Then, the language of $\mathcal{A}$ accepted from a tuple $(q_1, \ldots, q_n)$ of states is $\mathcal{L}^{\Box}(\mathcal{A})(q_1, \ldots, q_n) = \{t \in T^{\Box} \mid t(q_1, \ldots, q_n) \Longrightarrow q \text{ for some } q \in F\}$. Finally, we define the language accepted from a tuple of macro-states $(P_1, \ldots, P_n) \subseteq Q^n$ as the set $\mathcal{L}^{\Box}(\mathcal{A})(P_1, \ldots, P_n) = \bigcup\{\mathcal{L}^{\Box}(\mathcal{A})(q_1, \ldots, q_n) \mid (q_1, \ldots, q_n) \in P_1 \times \ldots \times P_n\}$. We define $Post_a(q_1, \ldots, q_n) := \{q \mid (q_1, \ldots, q_n) \xrightarrow{a} q\}$. For a tuple of macro-states, we let $Post_a(P_1, \ldots, P_n) := \bigcup\{Post_a(q_1, \ldots, q_n) \mid (q_1, \ldots, q_n) \in P_1 \times \cdots \times P_n\}$.

Let us use $t^{\Box}$ to denote the open tree that arises from a tree $t \in T(\Sigma)$ by replacing all the leaf symbols of t by $\Box$ and let for every leaf symbol $a \in \Sigma$, $I_a = \{q \mid \xrightarrow{a} q\}$ is the so called a-initial macro-state. Languages accepted *at* final states of $\mathcal{A}$ correspond to the languages accepted *from* tuples of initial macro-states of $\mathcal{A}$ as stated in Lemma 5.

Lemma 5. *Let t be a tree over Σ with leaves labeled by $a_1, \ldots, a_n$. Then $t \in \mathcal{L}(\mathcal{A})$ if and only if $t^{\Box} \in \mathcal{L}^{\Box}(\mathcal{A})(I_{a_1}, \ldots, I_{a_n})$.*

7.1 Upward Simulation

We now work towards defining suitable relations on states of TA allowing us to optimize the universality and inclusion checking. We extend relations $\preceq \in Q \times Q$ on states to tuples of states such that $(q_1, \ldots, q_n) \preceq (r_1, \ldots, r_n)$ iff $q_i \preceq r_i$ for each $1 \leq i \leq n$. We define

[1] Note that no internal nodes of an open tree can be labeled by $\Box$ as $\#(\Box) = 0$.

the set $\mathcal{A}^{\subseteq}$ of relations that imply inclusion of languages of tuples of states such that $\preceq \in \mathcal{A}^{\subseteq}$ iff $(q_1,\ldots,q_n) \preceq (r_1,\ldots,r_n)$ implies $\mathcal{L}^{\Box}(\mathcal{A})(q_1,\ldots,q_n) \subseteq \mathcal{L}^{\Box}(\mathcal{A})(r_1,\ldots,r_n)$.

We define an extension of simulation relations on states of word automata that satisfies the above property as follows. *Upward simulation* on $\mathcal{A}$ is a relation $\preceq \subseteq Q \times Q$ such that if $q \preceq r$, then (1) $q \in F \implies r \in F$ and (2) if $(q_1,\ldots,q_n) \xrightarrow{a} q'$ where $q = q_i$, then $(q_1,\ldots,q_{i-1},r,q_{i+1},\ldots,q_n) \xrightarrow{a} r'$ where $q' \preceq r'$. Upward simulations were discussed in [1], together with an efficient algorithm of computing them.[2]

Lemma 6. *For the maximal upward simulation $\preceq$ on $\mathcal{A}$, we have $\preceq \in \mathcal{A}^{\subseteq}$.*

The proof of this lemma can be obtained as follows. We first show that the maximal upward simulation $\preceq$ has the following property: If $(q_1,\ldots,q_n) \xrightarrow{a} q'$ in $\mathcal{A}$, then for every $(r_1,\ldots,r_n)$ with $(q_1,\ldots,q_n) \preceq (r_1,\ldots,r_n)$, there is $r' \in Q$ such that $q' \preceq r'$ and $(r_1,\ldots,r_n) \xrightarrow{a} r'$. From $(q_1,\ldots,q_n) \xrightarrow{a} q'$ and $q_1 \preceq r_1$, we have that there is some rule $(r_1,q_2,\ldots,q_n) \xrightarrow{a} s_1$ such that $q' \preceq s_1$. From the existence of $(r_1,q_2,\ldots,q_n) \xrightarrow{a} s_1$ and from $q_2 \preceq r_2$, we then get that there is some rule $(r_1,r_2,q_3,\ldots,q_n) \xrightarrow{a} s_2$ such that $s_1 \preceq s_2$, etc. Since the maximal upward simulation is transitive [1], we obtain the property mentioned above. This in turn implies Lemma 6.

7.2 Tree Automata Universality Checking

We now show how upward simulations can be used for optimized universality checking on tree automata. Let $\mathcal{A} = (\Sigma, Q, F, \Delta)$ be a tree automaton. We define $T_n^{\Box}(\Sigma)$ as the set of all open trees over Σ with n leaves. We say that an n-tuple $(q_1,\ldots,q_n)$ of states of $\mathcal{A}$ is universal if $\mathcal{L}^{\Box}(\mathcal{A})(q_1,\ldots,q_n) = T_n^{\Box}(\Sigma)$, this is, all open trees with n leaves constructible over Σ can be accepted from $(q_1,\ldots,q_n)$. A set of macro-states *MStates* is universal if all tuples in *MStates** are universal. From Lemma 5, we can deduce that $\mathcal{A}$ is universal (i.e., $\mathcal{L}(\mathcal{A}) = T(\Sigma)$) if and only if $\{I_a \mid a \in \Sigma_0\}$ is universal.

The following Lemma allows us to design a new TA universality checking algorithm in a similar manner to Algorithm 1 using Optimizations 1 and 2 from Section 3.

Lemma 7. *For any $\preceq \in \mathcal{A}^{\subseteq}$ and two tuples of macro-states of $\mathcal{A}$, we have $(R_1,\ldots,R_n) \preceq^{\forall\exists} (P_1,\ldots,P_n)$ implies $\mathcal{L}^{\Box}(\mathcal{A})(R_1,\ldots,R_n) \subseteq \mathcal{L}^{\Box}(\mathcal{A})(P_1,\ldots,P_n)$.*

Algorithm 3 describes our approach to checking universality of tree automata in pseudocode. It resembles closely Algorithm 1. There are two main differences: (1) The initial value of the *Next* set is the result of applying the function *Initialize* to the set $\{Minimize(I_a) \mid a \in \Sigma_0\}$. *Initialize* returns the set of all macro-states in $\{Minimize(I_a) \mid a \in \Sigma_0\}$, which are minimal w.r.t. $\preceq^{\forall\exists}$ (i.e., those macro states with the best chance of finding a counterexample to universality). (2) The computation of the *Post*-image of a set of macro-states is a bit more complicated. More precisely, for each symbol $a \in \Sigma_n, n \in \mathbb{N}$, we have to compute the post image of each n-tuple of macro-states from the set. We design the algorithm such that we avoid computing the *Post*-image of a tuple more than once. We define the *Post*-image $Post(MStates)(R)$ of a set of

[2] In [1], upward simulations are parameterized by some downward simulation. However, upward simulations parameterized by a downward simulation greater than the identity cannot be used in our framework since they do not generally imply inclusion of languages of tuples of states.

Algorithm 3: *Tree Automata Universality Checking*

Input: A tree automaton $\mathcal{A} = (\Sigma, Q, F, \Delta)$ and a relation $\preceq \in \mathcal{A}^{\subseteq}$.
Output: TRUE if $\mathcal{A}$ is universal. Otherwise, FALSE.

if $\exists a \in \Sigma_0$ *such that* I_a *is rejecting* **then return** *FALSE*;
Processed:=$\emptyset$;
Next:= *Initialize*$\{Minimize(I_a) \mid a \in \Sigma_0\}$;
while *Next* $\neq \emptyset$ **do**
    Pick and remove a macro-state R from *Next* and move it to *Processed*;
    foreach $P \in \{Minimize(R') \mid R' \in Post(Processed)(R)\}$ **do**
        if P *is a rejecting macro-state* **then return** *FALSE*;
        else if $\nexists Q \in Processed \cup Next$ *s.t.* $Q \preceq^{\forall\exists} P$ **then**
            Remove all Q from $Processed \cup Next$ s.t. $P \preceq^{\forall\exists} Q$;
            Add P to *Next*;
return *TRUE*

macro-states *MStates* w.r.t. a macro-states $R \in MStates$. It is the set of all macro-states $P = Post_a(P_1, \ldots, P_n)$ where $a \in \Sigma_n, n \in \mathbb{N}$ and R occurs at least once in the tuple $(P_1, \ldots, P_n) \in MStates^*$. Formally, $Post(MStates)(R) = \bigcup_{a \in \Sigma}\{Post_a(P_1, \ldots, P_n) \mid n = \#(a), P_1, \ldots, P_n \in MStates, R \in \{P_1, \ldots, P_n\}\}$.

The following theorem states correctness of Algorithm 3, which can be proved using similar invariants as in the case of Algorithm 1 when the notion of distance from an accepting state is suitably defined (see Appendix B for more details).

Theorem 2. *Algorithm 3 always terminates and returns* TRUE *if and only if the input tree automaton* $\mathcal{A}$ *is universal.*

7.3 Tree Automata Language Inclusion Checking

We are interested in testing language inclusion of two tree automata $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, F_{\mathcal{B}}, \Delta_{\mathcal{B}})$. From Lemma 5, we have that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff for every tuple $a_1, \ldots, a_n$ of symbols from Σ_0, $\mathcal{L}^{\Box}(\mathcal{A})(I^{\mathcal{A}}_{a_1}, \ldots, I^{\mathcal{A}}_{a_n}) \subseteq \mathcal{L}^{\Box}(\mathcal{B})(I^{\mathcal{B}}_{a_1}, \ldots, I^{\mathcal{B}}_{a_n})$. In other words, for any $a_1, \ldots, a_n \in \Sigma_0$, every open tree that can be accepted from a tuple of states from $I^{\mathcal{A}}_{a_1} \times \ldots \times I^{\mathcal{A}}_{a_n}$ can also be accepted from a tuple of states from $I^{\mathcal{B}}_{a_1} \times \ldots \times I^{\mathcal{B}}_{a_n}$. This justifies a similar use of the notion of product-states as in Section 5. We define the language of a tuple of product-states as $\mathcal{L}^{\Box}(\mathcal{A}, \mathcal{B})((q_1, P_1), \ldots, (q_n, P_n)) := \mathcal{L}^{\Box}(\mathcal{A})(q_1, \ldots, q_n) \setminus \mathcal{L}^{\Box}(\mathcal{B})(P_1, \ldots, P_n)$. Observe that we obtain that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff the language of every n-tuple (for any $n \in \mathbb{N}$) of product-states from the set $\{(i, I^{\mathcal{B}}_a) \mid a \in \Sigma_0, i \in I^{\mathcal{A}}_a\}$ is empty.

Our algorithm for testing language inclusion of tree automata will check whether it is possible to reach a product-state of the form (q, P) with $q \in F_{\mathcal{A}}$ and $P \cap F_{\mathcal{B}} = \emptyset$ (that we call accepting) from a tuple of product-states from $\{(i, I^{\mathcal{B}}_a) \mid a \in \Sigma_0, i \in I^{\mathcal{A}}_a\}$. The following lemma allows us to use Optimization 1(a) and Optimization 2 from Section 5.

Algorithm 4: *Tree Automata Language Inclusion Checking*

Input: TAs $\mathcal{A}$ and $\mathcal{B}$ over an alphabet Σ. A relation $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.

```
1  if there exists an accepting product-state in ⋃_{a∈Σ0} {(i, I_a^B) | i ∈ I_a^A} then return FALSE;
2  Processed := ∅;
3  Next := Initialize(⋃_{a∈Σ0} {(i, Minimize(I_a^B)) | i ∈ I_a^A});
4  while Next ≠ ∅ do
5      Pick and remove a product-state (r,R) from Next and move it to Processed;
6      foreach (p,P) ∈ {(r', Minimize(R')) | (r',R') ∈ Post(Processed)(r,R)} do
7          if (p,P) is an accepting product-state then return FALSE;
8          else if ∄p' ∈ P s.t. p ⪯ p' then
9              if ∄(q,Q) ∈ Processed ∪ Next s.t. p ⪯ q ∧ Q ⪯^{∀∃} P then
10                 Remove all (q,Q) from Processed ∪ Next s.t. q ⪯ p ∧ P ⪯^{∀∃} Q;
11                 Add (p,P) to Next;
12 return TRUE
```

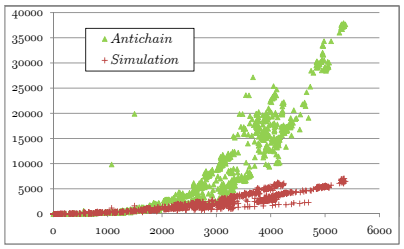
Lemma 8. *Let $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$. For any two tuples of states and two tuples of product-states such that $(p_1,\ldots,p_n) \preceq (r_1,\ldots,r_n)$ and $(R_1,\ldots,R_n) \preceq^{\forall\exists} (P_1,\ldots,P_n)$, we have $\mathcal{L}^{\square}(\mathcal{A},\mathcal{B})((p_1,P_1),\ldots,(p_n,P_n)) \subseteq \mathcal{L}^{\square}(\mathcal{A},\mathcal{B})((r_1,R_1),\ldots,(r_n,R_n))$.*

It is also possible to use Optimization 1(b) where we stop searching from product-states of the form (q,P) such that $q \preceq r$ for some $r \in P$. However, note that this optimization is of limited use for tree automata. Under the assumption that the automata $\mathcal{A}$ and $\mathcal{B}$ do not contain useless states, the reason is that for any $q \in Q_{\mathcal{A}}$ and $r \in Q_{\mathcal{B}}$, if q appears at a left-hand side of some rule of arity more than 1, then no reflexive relation from $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$ allows $q \preceq r$.[3]

Algorithm 4 describes our method for checking language inclusion of TA in pseudocode. It closely follows Algorithm 2. It differs in two main points. First, the initial value of the *Next* set is the result of applying the function *Initialize* on the set $\{(i, Minimize(I_a^{\mathcal{B}})) \mid a \in \Sigma_0, i \in I_a^{\mathcal{A}}\}$, where *Initialize* is the same function as in Algorithm 2. Second, the computation of the *Post* image of a set of product-states means that for each symbol $a \in \Sigma_n, n \in \mathbb{N}$, we construct the $Post_a$-image of each n-tuple of product-states from the set. Like in Algorithm 3, we design the algorithm such that we avoid computing the $Post_a$-image of a tuple more than once. We define the post image $Post(PStates)(r,R)$ of a set of product-states *PStates* w.r.t. a product-state $(r,R) \in PStates$. It is the set of all product-states (q,P) such that there is some $a \in \Sigma, \#(a) = n$ and some n-tuple $((q_1,P_1),\ldots,(q_n,P_n))$ of product-states from *PStates* that contains at least one occurrence of (r,R), where $q \in Post_a(q_1,\ldots,q_n)$ and $P = Post_a(P_1,\ldots,P_n)$.

Theorem 3. *Algorithm 4 always terminates and returns* TRUE *iff* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

[3] To see this, assume that an open tree t is accepted from $(q_1,\ldots,q_n) \in Q_{\mathcal{A}}^n, q = q_i, 1 \leq i \leq n$. If $q \preceq r$, then by the definition of $\preceq$, $t \in \mathcal{L}^{\square}(\mathcal{A} \cup \mathcal{B})(q_1,\ldots,q_{i-1},r,q_{i+1},\ldots,q_n)$. However, that cannot happen, as $\mathcal{A} \cup \mathcal{B}$ does not contain any rules with left hand sides containing both states from $\mathcal{A}$ and states from $\mathcal{B}$.



(a) Detailed results

Size	Antichain	Simulation
0-1000	0.059	0.099
1000-2000	1.0	0.7
2000-3000	3.6	1.69
3000-4000	11.2	3.2
4000-5000	20.1	4.79
5000-	33.7	6.3

(b) Average execution time for different NFA pair sizes (in seconds)

Fig. 3. Language inclusion checking on NFAs generated from a regular model checker

8 Experimental Results

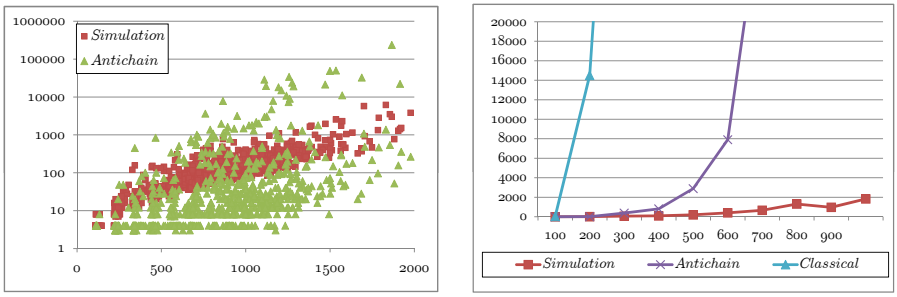
In this section, we describe our experimental results. We concentrated on experiments with inclusion checking, since it is more common than universality checking in various symbolic verification procedures, decision procedures, etc. We compared our approach, parameterized by maximal simulation (or, for tree automata, maximal upward simulation), with the previous pure antichain-based approach of [11], and with classical subset-construction-based approach. We implemented all the above in OCaml. We used the algorithm in [7] for computing maximal simulations. In order to make the figures easier to read, we often do not show the results of the classical algorithm, since in all of the experiments that we have done, the classical algorithm performed much worse than the other two approaches.

8.1 The Results on NFA

For language inclusion checking of NFA, we tested our approach on examples generated from the intermediate steps of a tool for abstract regular model checking [4]. In total, we have 1069 pairs of NFA generated from different verification tasks, which included verifying a version of the bakery algorithm, a system with a parameterized number of producers and consumers communicating through a double-ended queue, the bubble sort algorithm, an algorithm that reverses a circular list, and a Petri net model of the readers/writers protocol (cf. [4, 3] for a detailed description of the verification problems). In Fig. 3 (a), the horizontal axis is the sum of the sizes of the pairs of automata whose language inclusion we check, and the vertical axis is the execution time (the time for computing the maximal simulation is included). Each point denotes a result from inclusion testing for a pair of NFA. Fig. 3 (b) shows the average results for different NFA sizes. From the figure, one can see that our approach has a much better performance than the antichain-based one. Also, the difference between our approach and the antichain-based approach becomes larger when the size of the NFA pairs increases. If we compare the average results on the smallest 1000 NFA pairs, our approach is 60% slower than the the antichain-based approach. For the largest NFA pairs (those with size larger than 5000), our approach is 5.32 times faster than the the antichain-based approach.

We also tested our approach using NFA generated from random regular expressions. We have two different tests: (1) language inclusion does not always hold and (2) language inclusion always holds[4]. The result of the first test is in Fig. 4(a). In the figure,

[4] To get a sufficient number of tests for the second case, we generate two NFA $\mathcal{A}$ and $\mathcal{B}$ from random regular expressions, build their union automaton $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$, and test $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C})$.



(a) Language inclusion does not always hold (b) Language inclusion always holds

Fig. 4. Language inclusion checking on NFA generated from regular expressions

the horizontal axis is the sum of the sizes of the pairs of automata whose language inclusion we check, and the vertical axis is the execution time (the time for computing the maximal simulation is included). From Fig. 4(a), we can see that the performance of our approach is much more stable. It seldom produces extreme results. In all of the cases we tested, it always terminates within 10 seconds. In contrast, the antichain-based approach needs more than 100 seconds in the worst case. The result of the second test is in Fig. 4(b) where the horizontal axis is the length of the regular expression and the vertical axis is the average execution time of 30 cases in milliseconds. From Fig. 4(b), we observe that our approach has a much better performance than the antichain-based approach if the language inclusion holds. When the length of the regular expression is 900, our approach is almost 20 times faster than the antichain-based approach.

When the maximal simulation relation $\preceq$ is given, a natural way to accelerate the language inclusion checking is to use $\preceq$ to minimize the size of the two input automata by merging $\preceq$-equivalent states. In this case, the simulation relation becomes sparser. A question arises whether our approach has still a better performance than the antichain-based approach in this case. Therefore, we also evaluated our approach under this setting. Here again, we used the NFA pairs generated from abstract regular model checking [4]. The results show that although the antichain-based approach gains some speed-up when combined with minimization, it is still slower than our approach. The main reason is that in many cases, simulation holds only in one direction, but not in the other. Our approach can also utilize this type of relation. In contrast, the minimization algorithm merges only simulation equivalent states.

We have also evaluated the performance of our approach using backward language inclusion checking combined with maximal backward simulation. As Wulf et al. [11] have shown in their paper, backward language inclusion checking of two automata is in fact equivalent to the forward version on the reversed automata. This can be easily generalized to our case. The result is very consistent to what we have obtained; our algorithm is still significantly better than the antichain-based approach.

8.2 The Results on TA

For language inclusion checking on TA, we tested our approach on 86 tree automata pairs generated from the intermediate steps of a regular tree model checker [2] while verifying the algorithm of rebalancing red-black trees after insertion or deletion of a leaf

node. The results are given in Table 1. Our approach has a much better performance when the size of a TA pair is large. For TA pairs of size smaller than 200, our approach is on average 1.39 times faster than the antichain-based approach. However, for those of a size above 1000, our approach is on average 6.8 times faster than the antichain-based approach.

Size	Antichain (sec.)	Simulation (sec.)	Diff.	# of Pairs
0-200	1.05	0.75	139.5%	29
200-400	11.7	4.7	246%	15
400-600	65.2	19.9	327.9%	14
600-800	3019.2.6	568.7	531%	13
800-1000	4481.9	840.4	533%	5
1000-1200	11761.7	1720.9	683.4%	10

Table 1. Language inclusion checking on TA

9 Conclusion

We have introduced several original ways to combine simulation relations with antichains in order to optimize algorithms for checking universality and inclusion on NFA. We have also shown how the proposed techniques can be extended to the domain of tree automata. This was achieved by introducing the notion of languages of open trees accepted from tuples of tree automata states and using the maximal upward simulations parameterized by the identity proposed in our earlier work [1]. We have implemented the proposed techniques and performed a number of experiments showing that our techniques can provide a very significant improvement over currently known approaches. In the future, we would like to perform even more experiments, including, e.g., experiments where our techniques will be incorporated into the entire framework of abstract regular (tree) model checking or into some automata-based decision procedures. Apart from that, it is also interesting to develop the described techniques for other classes of automata (notably Büchi automata) and use them in a setting where the transitions of the automata are represented not explicitly but symbolically, e.g., using BDDs.

References

1. P.A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata. In *Proc. of TACAS'08*, LNCS 4963, 2008.
2. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-Based Universality and Inclusion Testing over Nondet. Finite Tree Automata. In *CIAA'08*, LNCS 5148, 2008.
3. A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *TACAS'05*, LNCS 3440, 2005.
4. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, LNCS 3114. Springer, 2004.
5. J. A. Brzozowski. Canonical Regular Expressions and Minimal State Graphs for Definite Events. In *Mathematical Theory of Automata*, 1962.
6. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *Proc. of CAV'92*, LNCS 575. Springer, 1992.
7. L. Holík and J. Šimáček. Optimizing an LTS-Simulation Algorithm. Technical Report FIT-TR-2009-03, Brno University of Technology, 2009.
8. J. E. Hopcroft. An *n.log n* Algorithm for Minimizing States in a Finite Automaton. Technical Report CS-TR-71-190, Stanford University, 1971.
9. A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Proc. of the 13th Annual Symposium on Switching and Automata Theory*. IEEE CS, 1972.
10. F. Møller. `http://www.brics.dk/automaton`, 2004.
11. M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV'06*, LNCS 4144. Springer, 2006.

A Correctness of the NFA Universality Checking

The following lemma is implied directly by the fact that if $\mathcal{L}(\mathcal{A})(P) \subseteq \mathcal{L}(\mathcal{A})(R)$, then the shortest word rejected by R is also rejected by P.

Lemma 9. *Let P and R be two macro-states such that $\mathcal{L}(\mathcal{A})(P) \subseteq \mathcal{L}(\mathcal{A})(R)$. We have $Dist(P) \leq Dist(R)$.*

Lemma 3. *The below two loop invariants hold in Algorithm 1:*

1. $\neg Univ(Processed \cup Next) \implies \neg Univ(\{I\})$.
2. $\neg Univ(\{I\}) \implies Dist(Processed) > Dist(Next)$.

Proof. It is trivial to see that the invariants hold at the entry of the loop, taking into account Lemma 2 covering the effect of the *Minimize* function. We show that the invariants continue to hold when the loop body is executed from a configuration of the algorithm in which the invariants hold. We use $Processed^{old}$ and $Next^{old}$ to denote the values of *Processed* and *Next* when the control is on line 4 before executing the loop body and we use $Processed^{new}$ and $Next^{new}$ to denote their values when the control gets back to line 4 after executing the loop body once. We assume that $Next^{old} \neq \emptyset$.

Let us start with Invariant 1. Assume first that $Univ(Processed^{old} \cup Next^{old})$ holds. Then, R must be universal, which holds also for all of its successors and, due to Lemma 2, also for their minimized versions, which may be added to *Next* on line 10. Hence, $Univ(Processed^{new} \cup Next^{new})$ holds after executing the loop body, and thus Invariant 1 holds too. Now assume that $\neg Univ(Processed^{old} \cup Next^{old})$ holds. Then, $\neg Univ(\{I\})$ holds, and hence Invariant 1 must hold for $Processed^{new}$ and $Next^{new}$ too.

We proceed to Invariant 2 and we assume that $\neg Univ(\{I\})$ holds (the other case being trivial). Hence, $Dist(Processed^{old}) > Dist(Next^{old})$ holds. We distinguish two cases:

1. $Dist(R) = \infty$ or $\exists Q \in Processed^{old} : Dist(Q) \leq Dist(R)$. In this case, $Dist(Processed)$ will not decrease on line 5. From $Dist(Processed^{old}) > Dist(Next^{old})$, there exists some macro-state R' in $Next^{old}$ s.t. $Dist(R') = Dist(Next^{old}) < Dist(Processed^{old}) \leq Dist(Q) \leq Dist(R)$. Therefore, $Dist(Next)$ will not change on line 5 either. Moreover, for any macro-state P, removing Q s.t. $P \preceq^{\forall\exists} Q$ from *Next* and *Processed* on line 9 and then adding P to *Next* on line 10 cannot invalidate $Dist(Processed^{new}) > Dist(Next^{new})$ since $Dist(P) \leq Dist(Q)$ due to Lemmas 2 and 9. Hence, Invariant 2 must hold for $Processed^{new}$ and $Next^{new}$ too.
2. $Dist(R) \neq \infty$ and $\neg\exists Q \in Processed^{old} : Dist(Q) \leq Dist(R)$. In this case, the value of $Dist(Processed)$ decreases to $Dist(R)$ on line 5. Clearly, $Dist(R) \neq 0$ or else we would have terminated before. Then there must be some successor R' of R which is either rejecting (and the loop stops without getting back to line 4) or one step closer to rejection, meaning that $Dist(R') < Dist(R)$. Moreover, R' either appears in $Next^{new}$ or there already exists some $R'' \in Next^{old}$ such that $R'' \preceq^{\forall\exists} R'$, meaning that $Dist(Processed^{new}) > Dist(Next^{new})$. It is impossible that $\exists R'' \in Processed^{old} : R'' \preceq^{\forall\exists} R'$, because $\forall R'' \in Processed^{old} : Dist(R'') > Dist(R) > Dist(R')$ and from Lemmas 2 and 9, $R'' \preceq^{\forall\exists} R'$ implies $Dist(R'') < Dist(R')$. Furthermore, if some macro-state is removed from *Processed* on line 9, $Dist(Processed)$ can only grow, and hence we are done. □

Lemma 10 (Termination). *Algorithm 1 eventually terminates.*

Proof. For the algorithm not to terminate, it would have to be the case that some macro-state is repeatedly added into *Next*. However, once some macro-state R is added into *Next*, there will always be some macro-state $Q \in Processed \cup Next$ such that $Q \preceq^{\forall\exists} R$. This holds since R either stays in *Next*, moves to *Procesed*, or is replaced by some Q such that $Q \preceq^{\forall\exists} R$ in each iteration of the loop. Hence, R cannot be added to *Next* for the second time since a macro-state is added to *Next* on line 10 only if there is no $Q \in Processed \cup Next$ such that $Q \preceq^{\forall\exists} R$. □

Theorem 1. *The algorithm terminates with the return value* FALSE *if the input automaton $\mathcal{A}$ is not universal. Otherwise, it terminates with the return value* TRUE.

Proof. From Lemma 10, the algorithm eventually terminates. It returns FALSE only if either the set of initial states is rejecting, or the minimized version R' of some successor S of a macro-state R chosen from *Next* on line 5 is found rejecting. In the latter case, due to Lemma 2, S is also rejecting. Then R is non-universal, and hence $Univ(Processed \cup Next)$ is false. By Lemma 3 (Invariant 1), we have $\mathcal{A}$ is not universal. The algorithm returns TRUE only when *Next* becomes empty. When *Next* is empty, $Dist(Processed) > Dist(Next)$ is not true. Therefore, by Lemma 3 (Invariant 2), $\mathcal{A}$ is universal. □

B Correctness of the TA Universality Checking

In this section, we prove correctness of Algorithm 3 in a very similar way to Algorithm 1, using suitably modified notions of distances and ranks. Let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a TA. For $n \geq 0$ and an n-tuple of macro-states $(Q_1, ..., Q_n)$ where $Q_i \subseteq Q$ for $1 \leq i \leq n$, we let $\mathbf{Dist}(Q_1, \ldots, Q_n) = 0$ iff $Q_i \cap F = \emptyset$ for some $i \in \{1, \ldots, n\}$. We define $\mathbf{Dist}(Q_1, \ldots, Q_n) = k \in \mathbb{N}^+ \cup \{\infty\}$ iff $Q_i \subseteq F$ for all $i \in \{1, \ldots, n\}$ and $k = min(\{|t| \mid t \in T_n^{\Box}(\Sigma) \wedge t \notin \mathcal{L}^{\Box}(\mathcal{A})(Q_1, \ldots, Q_n)\})$. Here, we assume $min(\emptyset) = \infty$. For a set *MStates* of macro-states over Q, we let $\mathbf{Rank}(MStates) = min(\{\mathbf{Dist}(Q_1, \ldots, Q_n) \mid n \geq 1 \wedge \forall 1 \leq i \leq n : Q_i \in MStates\})$ and we define $\mathbf{Univ}(MStates) \iff \mathbf{Rank}(MStates) = \infty$.

Lemma 11. *The below two loop invariants hold in Algorithm 3:*

1. $\neg\mathbf{Univ}(Processed \cup Next) \implies \neg\mathbf{Univ}(\{I_a \mid a \in \Sigma_0\})$.
2. $\neg\mathbf{Univ}(\{I_a \mid a \in \Sigma_0\}) \implies \mathbf{Rank}(Processed) > \mathbf{Rank}(Processed \cup Next)$.

Proof. It is trivial to see that the invariants hold at the entry of the loop, taking into account Lemma 7. We show that the invariants continue to hold when the loop body is executed from a configuration of the algorithm in which the invariants hold. We use $Processed^{old}$ and $Next^{old}$ to denote the values of *Processed* and *Next* when the control is on line 4 before executing the loop body and we use $Processed^{new}$ and $Next^{new}$ to denote their values when the control gets back to line 4 after executing the loop body once. We assume that $Next^{old} \neq \emptyset$.

Let us start with Invariant 1. Assume first that $\mathbf{Univ}(Processed^{old} \cup Next^{old})$ holds. Then, R can appear within tuples constructed over $Processed^{old} \cup Next^{old}$ which are universal only. In such a case, all macro-states Q reachable from all tuples T built over

$Processed^{old} \cup Next^{old}$ are such that when we add them to $Processed^{old} \cup Next^{old}$, the resulting set will still allow building universal tuples only. Otherwise, one could take a non-universal tuple containing some of the newly added macro-states Q, replace Q by the tuple T from which it arose, and obtain a non-universal tuple over $Processed^{old} \cup Next^{old}$, which is impossible. Hence, the possibility of adding the new macro-states to *Next* on line 10 cannot cause non-universality of $Processed^{new} \cup Next^{new}$, which due to Lemma 7 holds when adding the minimized macro-states too. Moreover, removing elements from *Next* or *Processed* cannot cause non-universality either. Hence, Invariant 1 holds over $Processed^{new}$ and $Next^{new}$ in this case. Next, let us assume that $\neg\mathbf{Univ}(Processed^{old} \cup Next^{old})$ holds. Then, $\neg\mathbf{Univ}(\{I_a \mid a \in \Sigma_0\})$ holds, and hence Invariant 1 must hold for $Processed^{new}$ and $Next^{new}$ too.

We proceed to Invariant 2 and we assume that $\neg\mathbf{Univ}(\{I_a \mid a \in \Sigma_0\})$ holds (the other case being trivial). Hence, $\mathbf{Rank}(Processed^{old}) > \mathbf{Rank}(Processed^{old} \cup Next^{old})$ holds. We distinguish two cases:

1. In order to build a tuple T over $Processed^{old}$ and $Next^{old}$ that is of **Dist** equal to $\mathbf{Rank}(Processed^{old} \cup Next^{old})$, one needs to use a macro-state Q in $Next^{old} \setminus \{R\}$. The macro-state Q stays in $Next^{new}$ or is replaced by a $\preceq^{\forall\exists}$-smaller macro-state added to *Next* on line 10 that, due to Lemma 7, can only allow to build tuples of the same or even smaller **Dist**. Likewise, the macro-states accompanying Q in T stay in $Next^{new}$ or $Processed^{new}$ or are replaced by $\preceq^{\forall\exists}$-smaller macro-states added to *Next* on line 10 allowing to build tuples of the same or smaller **Dist**, due to Lemma 7. Hence, moving R to *Processed* on line 5 cannot cause the invariant to break. Moreover, adding some further macro-states to *Next* on line 10 can only cause $\mathbf{Rank}(Processed \cup Next)$ to decrease while removing macro-states from *Processed* on line 9 can only cause $\mathbf{Rank}(Processed)$ to grow. Finally, replacing a macro-state in *Next* by a $\preceq^{\forall\exists}$-smaller one as a combined effect of lines 9 and 10 can again just decrease $\mathbf{Rank}(Processed \cup Next)$, due to Lemma 7. Hence, in this case, Invariant 2 must hold over $Processed^{new}$ and $Next^{new}$.
2. One can build some tuple T over $Processed^{old}$ and $Next^{old}$ that is of **Dist** equal to $\mathbf{Rank}(Processed^{old} \cup Next^{old})$ using $Processed^{old} \cup \{R\}$ only. In this case, there must be tuples constructible over $Processed^{old} \cup \{R\}$ and containing R that are not universal. We can distinguish the following subcases:
   (a) From some of the tuples built over $Processed^{old} \cup \{R\}$ and containing R, a non-accepting macro-state is reached via a single transition of $\mathcal{A}$, and the algorithm stops without getting back to line 4.
   (b) Otherwise, some of the macro-states that appear in $Post(Processed, R)$ and that will be added in the minimized form to *Next* must allow one to construct tuples which are of **Dist** smaller than those based on R. This holds since if a macro-state Q is reached from some tuple T containing R by a single transition, we can replace T in larger tuples leading to non-acceptation by Q, and hence decrease the size of the open tree needed to reach non-acceptation. Taking into account Lemma 7 to cover the effect of the minimization and using a similar reasoning as above for covering the effect of lines 9 and 10, it is then clear that Invariant 2 will remain to hold in this case.

□

Lemma 12. *Algorithm 3 eventually terminates.*

Proof. An analogy of the proof of Lemma 10. □

Theorem 2 can now be proved in a very similar way as Theorem 1.

C Correctness of the TA Language Inclusion Checking

We prove correctness of Algorithm 4 in a very similar way to Algorithm 2, using suitably modified notions of distances and ranks.

Let $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, F_{\mathcal{B}}, \Delta_{\mathcal{B}})$ be two tree automata. For $n \geq 0$ and an n-tuple of macro-states $((q_1, P_1), \ldots, (q_n, P_n))$, we let $\mathbf{Dist}((q_1, P_1), \ldots, (q_n, P_n)) = 0$ iff $\varepsilon \in \mathcal{L}^{\square}(\mathcal{A}, \mathcal{B})((q_1, P_1), \ldots, (q_n, P_n))$. Otherwise we define $\mathbf{Dist}((q_1, P_1), \ldots, (q_n, P_n)) = k \in \mathbb{N}^+ \cup \{\infty\}$ iff $k = \min(\{|t| \mid t \in T_n^{\square}(\Sigma) \wedge t \in \mathcal{L}^{\square}(\mathcal{A}, \mathcal{B})((q_1, P_1), \ldots, (q_n, P_n))\})$. Here, we assume $\min(\emptyset) = \infty$. For a set *PStates* of product-states, we let $\mathbf{Rank}(\mathit{PStates}) = \min(\{\mathbf{Dist}((q_1, P_1), \ldots, (q_n, P_n)) \mid n \geq 1 \wedge \forall 1 \leq i \leq n : (q_i, P_i) \in \mathit{PStates}\})$. The predicate $\mathit{Incl}(\mathit{PStates})$ is defined to be true iff $\mathbf{Rank}(\mathit{PStates}) = \infty$.

Lemma 13. *The following two loop invariants hold in Algorithm 4:*

1. $\neg \mathit{Incl}(\mathit{Processed} \cup \mathit{Next}) \implies \neg \mathit{Incl}(\bigcup_{a \in \Sigma_0} \{(i, I_a^{\mathcal{B}}) \mid i \in I_a^{\mathcal{A}}\})$.
2. $\neg \mathit{Incl}(\bigcup_{a \in \Sigma_0} \{(i, I_a^{\mathcal{B}}) \mid i \in I_a^{\mathcal{A}}\}) \implies \mathbf{Rank}(\mathit{Processed}) > \mathbf{Rank}(\mathit{Processed} \cup \mathit{Next})$.

The proof is similar to that of Lemma 11.

Lemma 14. *Algorithm 4 eventually terminates.*

Proof. An analogy of the proof of Lemma 10. □

Theorem 3 can now be proved in a very similar way as Theorem 1.