Antichain-based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata

FIT BUT Technical Report Series

Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar

Technical Report No. FIT-TR-2008-001
Faculty of Information Technology, Brno University of Technology

Last modified: May 8, 2008

Antichain-based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata

Ahmed Bouajjani[1], Peter Habermehl[1,2], Lukáš Holík[3], Tayssir Touili[1], and Tomáš Vojnar[3]

[1] LIAFA, CNRS and University Paris Diderot, France,
email: {abou,haberm,touili}@liafa.jussieu.fr
[2] LSV, ENS Cachan, CNRS, INRIA
[3] FIT, Brno University of Technology, Czech republic,
email: {holik,vojnar}@fit.vutbr.cz

Abstract. We propose new antichain-based algorithms for checking universality and inclusion of nondeterministic tree automata. We have implemented these algorithms in a prototype tool and we present experiments which show that the algorithms provide a significant improvement over the traditional determinisation-based approaches. Furthermore, we use the proposed antichain-based inclusion checking algorithm to build an abstract regular tree model checking framework based entirely on nondeterministic tree automata. We show the significantly improved efficiency of this framework on a series of experiments with verifying various programs over dynamic tree-shaped data structures linked by pointers.

1 Introduction

Tree automata have proved to be useful in numerous different areas, including, e.g., the implementation of decision procedures for various logics, XML manipulation, linguistics, or different areas of formal verifi cation of systems such as parameterised networks of processes, cryptographic protocols, or programs with dynamic linked data structures. A classical implementation of many of the operations—such as minimisation or inclusion checking—used for dealing with tree automata in the different application areas often assumes that the automata being handled are deterministic. However, like in our own practical experience discussed later in this paper, it may happen that the determinisation step yields automata which are too large to handle although the original nondeterministic automata are quite small. It may even be the case that the corresponding minimal deterministic automata are small, but one may not be able to compute them as the intermediary automata resulting from determinisation are too big.

As the situation is similar also for other kinds of automata, recently, a lot of research has been done to implement effi ciently operations like minimisation (or at least reduction), universality checking, or inclusion checking directly on nondeterministic word, Büchi, or tree automata. Here, we follow this line of research and propose and experimentally evaluate new *efficient algorithms* for *universality and inclusion checking* on *nondeterministic* (bottom-up) *tree automata*. Our work is based on using *antichains of sets of states* of the considered automata instead of the classical subset construction. In this way, we extend some of the antichain-based algorithms recently proposed for universality and inclusion checking over finite word automata [11] to tree automata (and we also show that the others are not practical for tree automata).

To evaluate the proposed algorithms, we have implemented them in a prototype tool over the Timbuk tree automata library [8] and tested them in a series of experiments showing that they provide a significant advantage over the traditional approaches based on determinisation. The experiments were done both on randomly generated automata with different densities of transitions and final states like in [11] as well as within an important complex application of tree automata. Indeed, our antichain-based inclusion checking algorithm for tree automata fills an important hole in the tree automata technology enabling us to implement an *abstract regular tree model checking* (ARTMC) framework based entirely on nondeterministic tree automata. ARTMC is a generic technique for an automated formal verification of various kinds of infinite-state and parameterised systems. In this paper, we, in particular, consider its use for verification of *programs manipulating dynamic tree-shaped data structures*, and we show that the use of nondeterministic tree automata instead of the deterministic ones gives a very significant improvement in the efficiency of the technique.

Related Work. In [11], antichains were used for dual forward and backward algorithms for universality and inclusion testing over finite word automata. In [7], antichains were applied for Büchi automata. Here, we show how the forward algorithms from [11] can be extended to finite (bottom-up) tree automata (in the form of algorithms computing upwards). We also show that the backward computation from word automata is not practical for tree automata (where it corresponds to a downward computation).

The regular tree model checking framework was studied in, e.g., [10, 5, 2], and abstract regular tree model checking in [3, 4]—in all cases using deterministic tree automata. When implementing a framework for abstract regular tree model checking based on nondeterministic tree automata, we exploit the recent results on simulation-based reduction of tree automata obtained in [1].

2 Preliminaries

We recall in this section basic definitions of terms, trees, and tree automata.

An alphabet Σ is ranked if it is endowed with a mapping $rank : \Sigma \to \mathbb{N}$. For $k \geq 0$, Σ_k is the set of elements of rank k. $\Sigma_k = \{f \in \Sigma \mid rank(f) = k\}$. Note that the sets Σ_k need not be disjoint. The elements of Σ_0 are called constants. The set T_Σ of terms over Σ is defined inductively as follows: if $f \in \Sigma_0$, then $f \in T_\Sigma$ and if $k \geq 1$, $f \in \Sigma_k$, and $t_1, \ldots, t_k \in T_\Sigma$, then $f(t_1, \ldots, t_k)$ is in T_Σ.

Definition 1. *A* **bottom-up tree automaton** *(we shall omit the 'bottom-up' below) is a tuple $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ where Q is a finite set of states, Σ is a ranked alphabet, $F \subseteq Q$ is a set of final states, and δ is a set of rules of the form (1) $f(q_1, \ldots, q_n) \to q$ or (2) $a(q_0) \to q$, where $a \in \Sigma_0$, $n \geq 1$, $f \in \Sigma_n$, and $q_1, \ldots, q_n, q, q' \in Q$.*

Let t be a term over Σ. A run of $\mathcal{A}$ on t can be done in a bottom-up manner as follows: first, we assign a state to each leaf according to the rules (2), then for each internal node, we must collect the states assigned to all its children and then associate a state to the node itself according to the rules (1). Formally, if during the state assignment process the subterms $t_1, \ldots, t_n$ are labelled with states $q_1, \ldots, q_n$, and if the rule $f(q_1, \ldots, q_n) \to q$ is in δ, which we will denote by $f(q_1, \ldots, q_n) \to_\delta q$ below, then the term $f(t_1, \ldots, t_n)$ is labelled with q. A term t is accepted if $\mathcal{A}$ reaches the root of t in a final state. The language accepted by the automaton $\mathcal{A}$ is the set of terms that it accepts: $\mathcal{L}(\mathcal{A}) = \{t \in T_\Sigma \mid t \xrightarrow{*}_\delta q(t) \text{ and } q \in F\}$.

We call a tree automaton *complete* if for all $n \geq 1$, $f \in \Sigma_n$, $q_1, ..., q_n \in Q$, there is at least one $q \in Q$ such that $f(q_1, ..., q_n) \to_\delta q$, and for each $a \in \Sigma_0$, there is at least one $q \in Q$ such that $a(q_0) \to_\delta q$. Note that a tree automaton may in general be *nondeterministic*—we call it *deterministic* if in both of the cases from the previous sentence, there is at most one right-hand side $q \in Q$.

3 Universality Checking

3.1 Lattices and Antichains

The following definitions are similar to the corresponding ones from [11]. Let Q be a finite set. An *antichain* over Q is a set $S \subseteq 2^Q$ such that $\forall s, s' \in S : s \not\subset s'$, i.e., a set of pairwise incomparable subsets of Q. We denote by L the set of antichains. A set $s \in S \subseteq 2^Q$ is minimal in S iff $\forall s' \in S : s' \not\subset s$. Given a set $S \subseteq 2^Q$, we denote by $\lfloor S \rfloor$ the set of minimal elements of S. We define a partial order on antichains: for two antichains $S, S' \in L$, let $S \sqsubseteq S'$ iff $\forall s' \in S' \; \exists s \in S : s \subseteq s'$. Given two antichains $S, S' \in L$, the $\sqsubseteq$-lub (least upper bound) is the antichain $S \sqcup S' = \lfloor \{s \cup s' | s \in S \wedge s' \in S'\} \rfloor$ and the $\sqsubseteq$-glb (greatest lower bound) is the antichain $S \sqcap S' = \lfloor \{s | s \in S \vee s \in S'\rfloor$. These definitions can be extended to lub and glb of arbitrary subsets of L in the obvious way, yielding the operators $\bigsqcup$ and $\bigsqcap$. Then, it is easy to see that we obtain a complete lattice $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \{\emptyset\}, \emptyset)$, where $\{\emptyset\}$ is the minimal element and $\emptyset$ the maximal element.

3.2 Upward Universality Checking Using Antichains

To check for universality of a tree automaton, the standard approach consists in making the automaton complete, determinising it, complementing, and checking for emptiness. Since determinisation is an expensive step, we propose in this section an algorithm that allows to check universality for tree automata without determinisation. The idea of our technique is to find at least one term that is not accepted by the automaton. To do this, we perform a kind of symbolic simulation of the automaton to cover all runs that necessarily lead to non-accepting states.

In what follows, $q, q_1, q_2, ...$ denote states of nondeterministic tree automata, $s, s_1, s_2, ...$ denote sets of such states, and $S, S_1, S_2, ...$ denote antichains of sets of states.

We first give some definitions. We suppose our automata to be complete. Let $f \in \Sigma_n$, then $Post_f^\delta(s_1, ..., s_n) = \{q \mid \exists q_i \in s_i, 1 \leq i \leq n : f(q_1, ..., q_n) \to_\delta q\}$. We omit δ if it is understood from the context. Let us define $Post(S) = \lfloor \{Post_f(s_1, ..., s_n) \mid n \in \mathbb{N}, s_1, ..., s_n \in S, f \in \Sigma_n\} \rfloor$. Clearly, $Post$ is monotonic wrt. $\sqsubseteq$. Let $Post_0(S) =$

S and $Post_i(S) = Post(Post_{i-1}(S)) \sqcap S$ for all $i > 0$. As $Post$ is monotonic, we have

$$\forall S \in L \; \forall i \geq 0 : Post_{i+1}(S) \sqsubseteq Post_i(S) \tag{1}$$

Since we work on a finite lattice, this implies that for all S there exists j_S such that $Post_{j_S}(S) = Post_{j_S+1}(S)$. We let $Post^*(S) = Post_{j_S}(S)$.

Lemma 1. *Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ be an automaton and t a term over Σ. Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$, then $Post^*(\{\{q_0\}\}) \sqsubseteq \{s\}$.*

Proof. We proceed by structural induction on t.

Basic case. $t = a \in \Sigma_0$. Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Then $s = \{q \mid a(q_0) \rightarrow_\delta q\}$ which is equal to $Post_a(\{q_0\})$. Therefore, there exists $s' \in Post(\{\{q_0\}\})$ s.t. $s' \subseteq s$ since $Post$ is obtained by taking the minimal elements. Furthermore, because of (1), there is also $s'' \subseteq s'$ such that $s'' \in Post^*(\{\{q_0\}\})$.

Induction step. $t = f(t_1, ..., t_n)$. Let $s_i = \{q \in Q \mid t_i \xrightarrow{*}_\delta q\}$ for $i \in \{1, ..., n\}$. Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Then, clearly, $s = \{q \mid \exists q_1 \in s_1, ..., q_n \in s_n : f(q_1, ..., q_n) \rightarrow_\delta q\}$. By induction, it follows that there exists $s'_i \subseteq s_i$ s.t. $s'_i \in Post^*(\{\{q_0\}\})$. Let $s' = Post_f(s'_1, ..., s'_n)$. Then, it is clear by the definition of $Post_f$ that we have $s' \subseteq s$, and by definition of $Post^*$, there exists $s'' \subseteq s'$ such that $s'' \in Post^*(\{\{q_0\}\})$. □

Lemma 2. *Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ be an automaton and let $s \in Post^*(\{\{q_0\}\})$. Then there exists a term t over Σ such that $s = \{q \mid t \xrightarrow{*}_\delta q\}$.*

Proof. Let i be the minimum index such that $s \in Post_i(\{\{q_0\}\})$. We proceed by induction on i.

Basic case. $i = 1$. Then there exists a symbol $a \in \Sigma_0$ such that $s \in Post_a(\{q_0\})$. It is clear by the definition of $Post_a$ that $s = \{q \mid a \xrightarrow{*}_\delta q\}$.

Induction step. Let $i > 1$. Then, there exists $f \in \Sigma_n$ and $s_1, ..., s_n \in Post_{i-1}(\{\{q_0\}\})$ with $s = Post_f(s_1, ..., s_n)$. By induction, there exists $t_1, ..., t_n$ s.t. for $j \in \{1, ..., n\}$, $s_j = \{q \mid t_j \xrightarrow{*}_\delta q\}$. Let $t = f(t_1, ...t_n)$. By definition of $Post_f$, $s = \{q | t \xrightarrow{*}_\delta q\}$. □

We can now give a theorem allowing to decide universality *without determinisation*.

Theorem 1. *A tree automaton $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ is not universal if and only if $\exists s \in Post^*(\{\{q_0\}\}) : s \subseteq \overline{F}$.*

Proof. Suppose $\mathcal{A}$ is not universal. Let then t be a term that is not accepted by $\mathcal{A}$. Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Since t is not accepted by the automaton, $s \subseteq \overline{F}$. It follows by Lemma 1 that there is $s' \in Post^*(\{\{q_0\}\})$ such that $s' \subseteq s \subseteq \overline{F}$.

Suppose that there exists $s \in Post^*(\{\{q_0\}\})$ such that $s \subseteq \overline{F}$. By Lemma 2, there exists a term t such that $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Since $s \subseteq \overline{F}$, t is not accepted by $\mathcal{A}$. □

3.3 Experiments with Upward Universality Checking Using Antichains

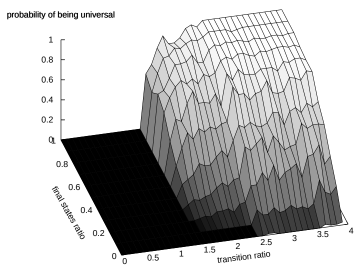
We have implemented the above approach for testing universality of tree automata in a simple prototype based on the Timbuk tree automata library [8]. The results we obtained from experimenting with the implementation are given in Fig. 1. We ran our tests on randomly generated automata and on automata obtained from abstract regular tree model checking applied in verification of several pointer-manipulating programs. All the experiments were run on an Intel Xeon processor at 2.7GHz with 16GB of memory.

In the random tests, we first used automata with 20 states and we varied the *density of their transitions* (the average number of different right-hand side states for a given left-hand side of a transition rule, i.e., $|\delta|/|\{f(q_1,...,q_n) \mid \exists q \in Q : f(q_1,...,q_n) \rightarrow_\delta q\}|$) and the *density of their final states* (i.e., $|F|/|Q|$). Fig. 1(a) shows the probability of such automata being universal, and Fig. 1(b) the average times needed for checking the universality of such automata using our antichain-based approach. The most difficult instances are naturally those where the probability of being universal is about one half. In Fig. 1(c), we show how the running times change for some selected instances of the problem (in terms of some selected densities of transitions and final states, including those for which the problem is the most difficult) when the number of states of the automata grows. The figure also shows the time needed when the universality is checked using determinisation, complement, and emptiness checking. We see that the antichain-based approach behaves in a significantly better way. The same conclusion can then be drawn also from the results shown in Fig. 1(d) obtained on automata from experimenting with abstract regular tree model checking applied for verifying various real-life procedures manipulating trees, which are presented in Section 5.3.

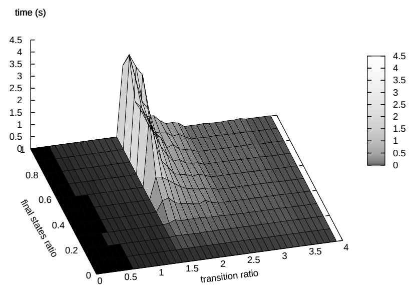
3.4 Downward Universality Checking with Antichains

The *upward universality checking* that we have introduced above for tree automata conceptually corresponds to the *forward* universality checking of finite word automata discussed in [11]. In [11], a dual *backward* universality checking based on computing the *controllable predecessors* of the set of non-final states is also introduced. Here, the controllable predecessors are the predecessors that can be forced by some input symbol to continue into a given set of states. Then, the automaton is non-universal iff the set of initial states is covered by the controllable predecessors of the non-final states.

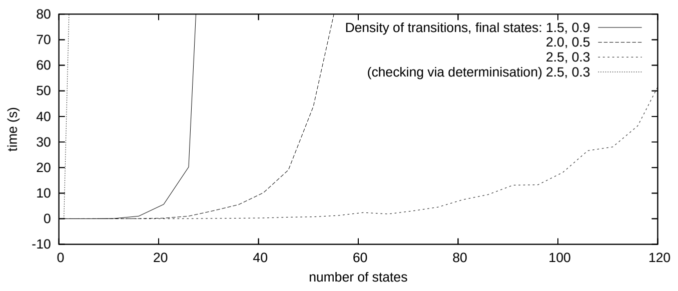
Downward universality checking for tree automata as a dual approach to the upward universality checking is problematic. This is because the set of controllable predecessors of a set of states $s \subseteq Q$ of a tree automaton $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ is not a set of states, but a set of tuples of states. Namely, $CPre(s) = \{(q_1,...,q_n) \mid n \in \mathbb{N} \wedge \exists f \in \Sigma\ \forall q \in Q : f(q_1,...,q_n) \rightarrow_\delta q \Rightarrow q \in s\}$. Note that if we flatten the set $CPre(s)$ to the set $FCPre(s)$ of states that appear in some of the tuples of $CPre(s)$ and check that starting from q_0 the computation can be forced into some subset of $FCPre(s)$, then this does not imply that the computation can be forced into some state from s. This is the case because for any rule $f(q_1,...,q_n) \rightarrow_\delta q$, $q \in s$, not all of the states q_1, ..., q_n may be reached. Furthermore, it is too strong to require that starting from q_0, it must be possible to force the computation into all states of $FCPre_f(s)$. Clearly, it is enough if the computation starting from q_0 can be forced into s via some of the vectors



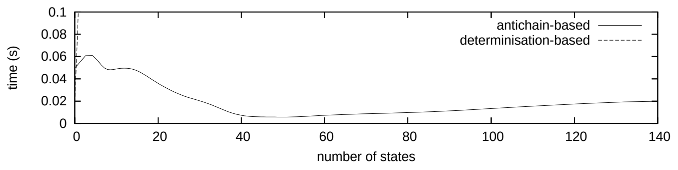
(a) Probability that a tree automaton (TA) with 20 states and some density of transitions and final states is universal



(b) Average times of antichain-based universality checking on TA with 20 states and some density of transitions and final states



(c) Determinisation-based and antichain-based universality checking on TA with selected densities of transitions and final states



(d) Determinisation-based and antichain-based universality checking on TA from abstract regular tree model checking

Fig. 1. Experiments with universality checking on tree automata

in $CPre(s)$, and not necessarily all of them. Also, if we keep $CPre(s)$ for $s \subseteq Q$ as a set of vectors, we have to subsequently defi ne the notion of a controllable predecessor for sets of vectors of states, which is a set of vectors of vectors of states, and so on. Clearly, such an approach is not practical.

4 Inclusion Checking

Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', q'_0, \delta')$ be two tree automata. We want to check whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. The traditional approach consists in computing the complement of $\mathcal{B}$, and check whether it has an empty intersection with $\mathcal{A}$. This is expensive because computing the complement necessitates determinisation. We show in this section how to check inclusion without determinisation.

As previously, the idea is to fi nd at least one term that is accepted by $\mathcal{A}$ and not by $\mathcal{B}$. For that, we simultaneously simulate the runs of the two automata by dealing with pairs of the form (p, s) such that $p \in Q$ and $s \subseteq Q'$, where the fi rst component memorises the run of $\mathcal{A}$ and the second component all the possible runs of $\mathcal{B}$. Then, if t is a term accepted by $\mathcal{A}$ and not by $\mathcal{B}$, this simultaneous run of the two automata reaches the root of t at a pair of the form (p, s) with $p \in F$ and $s \subseteq \overline{F'}$. Notice that s must represent *all* the possible runs of $\mathcal{B}$ on t in order to be sure that no run of $\mathcal{B}$ can accept the term t. This is the reason why we need s to be a set of states.

Formally, an *antichain* over $Q \times 2^{Q'}$ is a set $\mathcal{S} \subseteq Q \times 2^{Q'}$ such that for every $(p, s), (p', s') \in \mathcal{S}$, if $p = p'$, then s and s' are incomparable ($s \not\subseteq s'$ and $s' \not\subseteq s$). We denote by L_I the set of all antichains over $Q \times 2^{Q'}$. Given a set $\mathcal{S} \in Q \times 2^{Q'}$, an element $(p, s) \in \mathcal{S}$ is *minimal* if for every $s' \subset s$, $(p, s') \notin \mathcal{S}$. As previously, we denote by $\lfloor \mathcal{S} \rfloor$ the set of minimal elements of $\mathcal{S}$. Given two antichains $\mathcal{S}$ and $\mathcal{S}'$, we defi ne the order $\sqsubseteq_I$, the least upper bound $\sqcup_I$, and the greatest lower bound $\sqcap_I$ as follows: $\mathcal{S} \sqsubseteq_I \mathcal{S}'$ iff for every $(p, s') \in \mathcal{S}'$, there exists $(p, s) \in \mathcal{S}$ such that $s \subseteq s'$; $\mathcal{S} \sqcup_I \mathcal{S}' = \lfloor \{(p, s \cup s') \mid (p, s) \in \mathcal{S} \wedge (p, s') \in \mathcal{S}'\} \rfloor$; and $\mathcal{S} \sqcap_I \mathcal{S}' = \lfloor \{(p, s) \mid (p, s) \in \mathcal{S} \vee (p, s) \in \mathcal{S}'\} \rfloor$.

These defi nitions can be extended to arbitrary sets in the standard way leading to the operators $\bigsqcup_I$ and $\bigsqcap_I$. This defi nes a complete lattice as in Section 3.1.

For a given $f \in \Sigma_n$, we defi ne $IPost_f\big((p_1, s_1), ..., (p_n, s_n)\big) = \{(p, s) \mid f(p_1, ..., p_n) \rightarrow_\delta p \wedge s = Post_f^{\delta'}(s_1, ..., s_n)\}$. Let $\mathcal{S}$ be an antichain over $Q \times 2^{Q'}$. Then, we let $IPost(\mathcal{S}) = \lfloor \{IPost_f\big((p_1, s_1), \ldots, (p_n, s_n)\big) \mid n \in \mathbb{N}, (p_1, s_1), ..., (p_n, s_n) \in \mathcal{S}, f \in \Sigma_n\} \rfloor$. Let $IPost_0(\mathcal{S}) = \mathcal{S}$ and $IPost_i(\mathcal{S}) = IPost\big(IPost_{i-1}(\mathcal{S})\big) \sqcap_I \mathcal{S}$. As before, we can show that $\forall S \in L_I\ \forall i \geq 0 : IPost_{i+1}(S) \sqsubseteq_I IPost_i(S)$, and that for every antichain $\mathcal{S}$, there exists a J such that $IPost_{J+1}(\mathcal{S}) = IPost_J(\mathcal{S})$. Let $IPost^*(\mathcal{S}) = IPost_J(\mathcal{S})$. Then, we can show the following lemma. The proof is similar to the one of Lemma 1.

Lemma 3. *Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', q'_0, \delta')$ be two automata, and let t be a term over Σ. Let $p \in Q$ such that $t \xrightarrow{*}_\delta p$, and $s = \{q \in Q' \mid t \xrightarrow{*}_{\delta'} q\}$. Then, $IPost^*\big(\{(q_0, \{q'_0\})\}\big)\big) \sqsubseteq_I \{(p, s)\}$.*

We can also show the following lemma. Its proof is similar to the one of Lemma 2.

Lemma 4. *Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', q'_0, \delta')$ be two automata, and let $(p, s) \in IPost^*(\{(q_0, \{q'_0\})\})$. Then there exists a term t over Σ such that $t \xrightarrow{*}_\delta p$ and $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$.*

Then, we can decide inclusion *without determinising the automata* as follows:

Theorem 2. *Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', q'_0, \delta')$ be two automata. Then, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff for every $(p, s) \in IPost^*(\{(q_0, \{q'_0\})\})$, $p \in F \Rightarrow s \not\subseteq \overline{F'}$.*

Proof. Suppose there exists $(p, s) \in IPost^*(\{(q_0, \{q'_0\})\})$ such that $p \in F$ and $s \subseteq \overline{F'}$. Lemma 4 implies that there exists a term t such that $t \xrightarrow{*}_\delta p$ and $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$. Since $p \in F$ and $s \subseteq \overline{F'}$, t is accepted by $\mathcal{A}$ and not by $\mathcal{B}$, i.e., $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$.

Suppose now that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$. Let then t be a term accepted by $\mathcal{A}$ and not by $\mathcal{B}$. Let $p \in F$ such that $t \xrightarrow{*}_\delta p$, and let $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$. We have that $s \subseteq \overline{F'}$. Lemma 3 implies that $IPost^*\big(\{(q_0, \{q'_0\})\}\big)$ contains a pair (p, s') such that $s' \subseteq s \subseteq \overline{F'}$. $\square$

4.1 Experiments with Inclusion Checking Using Antichains

Below, in Fig. 2 and Fig. 3, we present the results that we have obtained from experimenting with our prototype implementation of the antichain-based inclusion checking for tree automata, which we have built on top of the Timbuk tree automata library. The experiments were performed on an Intel Xeon processor at 2.7GHz with 16GB of available memory (the same as in Section 3.3).

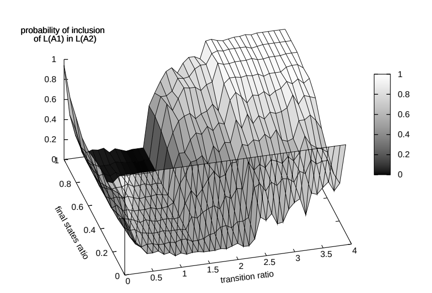
We first ran our tests on pairs of randomly generated automata having 10 states and different possible densities of transitions and final states. The probability that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ holds for randomly generated tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (both having the same densities of transitions and final states) is shown in Fig. 2(a). Fig. 2(b) then shows how the antichain-based inclusion checking behaves on such automata. We see that its time consumption is naturally growing for automata where the probability of whether $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ holds is neither too low nor too high.

Fig. 2(c) and Fig. 2(d) show what happens if either $\mathcal{A}_1$ or $\mathcal{A}_2$ is left completely random, and only $\mathcal{A}_2$ or $\mathcal{A}_1$, respectively, follows a given density of transitions and final states. The fact that the results in Fig. 2(c) follow Fig. 2(b), whereas the time consumption in Fig. 2(d) is roughly implied by the size of $\mathcal{A}_1$ (in terms of transitions), implies that the time consumption of the antichain-based inclusion checking is—as expected—influenced much more by the automaton $\mathcal{A}_2$.

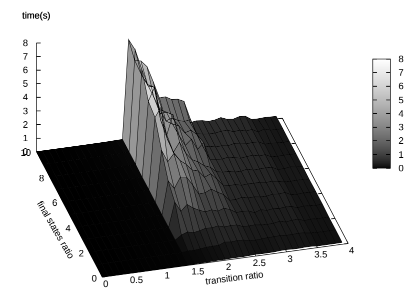
Finally, in Fig. 3(a), we show how the running times change for some selected instances of the problem (in terms of some selected densities of transitions and final states, including those for which the problem is the most difficult) when the number of states of the automata starts growing. The figure also shows the time needed when the inclusion checking is based on determinising and complementing $\mathcal{A}_2$ and checking emptiness of the language $\mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)}$. We see that the antichain-based approach really behaves in a very significantly better way. The same conclusion can then be drawn also from the results shown in Fig. 3(b) that we obtained on automata saved from experimenting with abstract regular tree model checking applied for verifying various real-life procedures manipulating trees (cf. Section 5.3). In fact, the antichain-based inclusion checking allowed us to implement an abstract regular tree model checking framework entirely based on nondeterministic tree automata which is significantly more efficient than the framework based on deterministic automata.

5 Regular Tree Model Checking

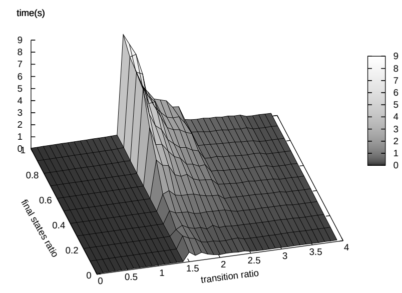
Regular tree model checking (RTMC) [10, 5, 2, 3] is a general and uniform framework for verifying infinite-state systems. In RTMC, configurations of a system being verified



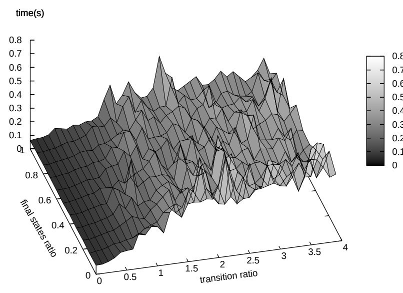
(a) Probability of $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ for tree automata (TA) with 10 states and some density of transitions and final states



(b) Average times of antichain-based inclusion checking on TA with some density of transitions and final states



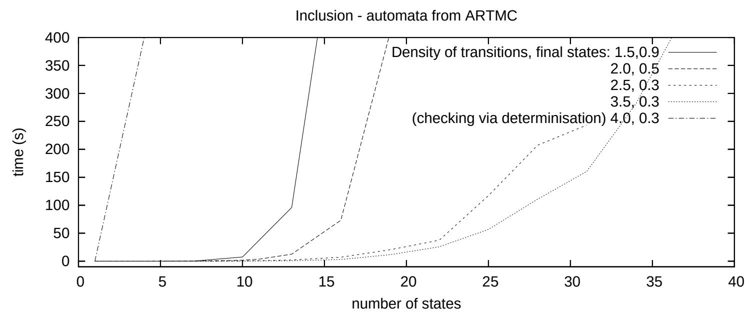
(c) Antichain-based inclusion checking on TA, $\mathcal{A}_1$ random, $\mathcal{A}_2$ with some density of transitions and final states



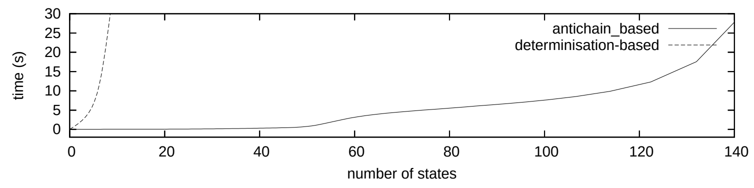
(d) Antichain-based inclusion checking on TA, $\mathcal{A}_2$ random, $\mathcal{A}_1$ with some density of transitions and final states

Fig. 2. Experiments with inclusion checking on tree automata

are encoded by trees, sets of the configurations by tree automata, and transitions of the verified system by a term rewriting system (usually given as a tree transducer or a set of tree transducers). Then, verification problems based on performing reachability analysis correspond to computing closures of regular languages under rewriting systems, i.e., given a term rewriting system τ and a regular tree language I, one needs to compute $\tau^*(I)$, where τ^* is the reflexive-transitive closure of τ. This computation is impossible in general. Therefore, the main issue in RTMC is to find accurate and powerful fixpoint acceleration techniques helping the convergence of computing language closures. One of the most successful acceleration techniques used in RTMC is abstraction whose use leads to the so-called *abstract regular tree model checking* (ARTMC) [3], on which we concentrate in this work.



(a) Determinisation-based and antichain-based inclusion checking on TA with selected densities of transitions and final states



(b) Determinisation-based and antichain-based inclusion checking on TA from abstract regular tree model checking

Fig. 3. Further experiments with inclusion checking on tree automata

5.1 Abstract Regular Tree Model Checking

We now briefly recall the basic principles of ARTMC in the way they were introduced in [3]. Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. Let $\mathcal{I} \in \mathbb{M}_\Sigma$ be a tree automaton describing a set of initial confi gurations, τ a term rewriting system describing the behaviour of a system, and $\mathcal{B} \in \mathbb{M}_\Sigma$ a tree automaton describing a set of bad confi gurations. The safety verifi cation problem can now be formulated as checking whether the following holds:

$$\tau^*(\mathcal{L}(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) = \emptyset \tag{2}$$

In ARTMC, the precise set of reachable confi gurations $\tau^*(\mathcal{L}(\mathcal{I}))$ is not computed to solve Problem (2). Instead, its overapproximation is computed by interleaving the application of τ and the union in $\mathcal{L}(\mathcal{I}) \cup \tau(\mathcal{L}(\mathcal{I})) \cup \tau(\tau(\mathcal{L}(\mathcal{I}))) \cup ...$ with an application of an abstraction function α. The abstraction is applied on the tree automata encoding the so-far computed sets of reachable confi gurations.

An abstraction function is defi ned as a mapping $\alpha : \mathbb{M}_\Sigma \rightarrow \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \mathcal{L}(\alpha'(\mathcal{A})) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. Given a term rewriting system τ

and an abstraction α, a mapping $\tau_\alpha : \mathbb{M}_\Sigma \rightarrow \mathbb{M}_\Sigma$ is defined as $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \tau_\alpha(\mathcal{A}) = \hat{\tau}(\alpha(\mathcal{A}))$ where $\hat{\tau}(\mathcal{A})$ is the minimal deterministic automaton describing the language $\tau(\mathcal{L}(\mathcal{A}))$. An abstraction α is *finitary*, if the set $\mathbb{A}_\Sigma$ is finite.

For a given abstraction function α, one can compute iteratively the sequence of automata $(\tau_\alpha^i(\mathcal{I}))_{i\geq 0}$. If the abstraction α is finitary, then there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(\mathcal{I}) = \tau_\alpha^k(\mathcal{I})$. The definition of the abstraction function α implies that $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \supseteq \tau^*(\mathcal{L}(\mathcal{I}))$.

If $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$, then Problem (2) has a positive answer. If the intersection is non-empty, one must check whether a real or a spurious counterexample has been encountered. The spurious counterexample may be caused by the used abstraction (the counterexample is not reachable from the set of initial configurations). Assume that $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$, which means that there is a symbolic path:

$$\mathcal{I},\ \tau_\alpha(\mathcal{I}),\ \tau_\alpha^2(\mathcal{I}), ..., \tau_\alpha^{n-1}(\mathcal{I}),\ \tau_\alpha^n(\mathcal{I}) \tag{3}$$

such that $\mathcal{L}(\tau_\alpha^n(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$.

Let $X_n = \mathcal{L}(\tau_\alpha^n(\mathcal{I})) \cap \mathcal{L}(\mathcal{B})$. Now, for each l, $0 \leq l < n$, $X_l = \mathcal{L}(\tau_\alpha^l(\mathcal{I})) \cap \tau^{-1}(X_{l+1})$ is computed. Two possibilities may occur: (a) $X_0 \neq \emptyset$, which means that Problem (2) has a negative answer, and $X_0 \subseteq \mathcal{L}(\mathcal{I})$ is a set of dangerous initial configurations. (b) $\exists m, 0 \leq m < n, X_{m+1} \neq \emptyset \wedge X_m = \emptyset$ meaning that the abstraction function is too rough—one needs to refine it and start the verification process again.

In [3], two general-purpose kinds of abstractions are proposed. Both are based on *automata state equivalences*. Tree automata states are split into several equivalence classes, and all states from one class are collapsed into one state. An abstraction becomes finitary if the number of equivalence classes is finite. The refinement is done by refining the equivalence classes. Both of the proposed abstractions allow for an automatic refinement to exclude the encountered spurious counterexample.

The first proposed abstraction is an *abstraction based on languages of trees of a finite height*. It defines two states equivalent if their languages up to the give height n are equivalent. There is just a finite number of languages of height n, therefore this abstraction is finitary. A refinement is done by an increase of the height n. The second proposed abstraction is an *abstraction based on predicate languages*. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent if the languages $\mathcal{L}(\mathcal{A}_{q_1})$ and $\mathcal{L}(\mathcal{A}_{q_2})$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$ provided that $\mathcal{A}_{q_1} = (Q, \Sigma, F, q_1, \delta)$ and $\mathcal{A}_{q_2} = (Q, \Sigma, F, q_2, \delta)$. Since there is just a finite number of subsets of $\mathcal{P}$, the abstraction is finitary. A refinement is done by adding new predicates, i.e. tree automata corresponding to the languages of all the states in the automaton of X_{m+1} from the analysis of spurious counterexample ($X_m = \emptyset$).

5.2 Nondeterministic Abstract Regular Tree Model Checking

As is clear from the above mentioned definition of $\hat{\tau}$, ARTMC was originally defined for and tested on *minimal deterministic* tree automata (DTA). However, the various experiments done showed that the determinisation step is a significant bottleneck. To avoid it and to implement ARTMC using nondeterministic tree automata (NTA), we need the following operations over NTA: (1) application of the transition relation τ, (2) union, (3) abstraction and its refinement, (4) intersection with the set of bad configurations, (5) emptiness, and (6) inclusion checking (needed for testing if the abstract reachability computation has reached a fixpoint). Finally, (7) a method to reduce the size of the computed NTA is also desirable—$\hat{\tau}(\mathcal{A})$ is then redefined to be the reduced version of the NTA obtained from an application of τ on an NTA $\mathcal{A}$.

An implementation of Points (1), (2), (4), and (5) is easy. Moreover, concerning Point (3), the abstraction mechanisms of [3] can be lifted to work on NTA in a straightforward way while preserving their guarantees to be finitary, overapproximating, and the ability to exclude spurious counterexamples. Furthermore, the recent work [1] gives efficient algorithms for reducing NTA based on computing suitable simulation equivalences on their states, which covers Point (7). Hence, the last obstacle for implementing nondeterministic ARTMC was Point (6), i.e., the need to efficiently check inclusion on NTA. We have solved this problem by the approach proposed in Section 4, which allowed us to implement a nondeterministic ARTMC framework in a prototype tool and test it on suitable examples. Below, we present the first very encouraging results that we have achieved.

5.3 Experiments with Nondeterministic ARTMC

We have implemented the nondeterministic ARTMC framework using the Timbuk tree library [8] and compared it with an ARTMC implementation based on the same library, but using DTA. In particular, the deterministic ARTMC framework uses determinisation and minimisation after computing the effect of each forward or backward step to try to keep the automata as small as possible and to allow for easy fixpoint checking: The fixpoint checking on DTA is not based on inclusion, but identity checking on the obtained automata (due to the fact that the computed sets are only growing and minimal DTA are canonical). For NTA, the tree automata reduction from [1] that we use does not yield canonical automata, and so the antichain-based inclusion checking is really needed.

We have applied the framework to verify several procedures manipulating dynamic tree-shaped data structures linked by pointers. The trees being manipulated are encoded directly as the trees handled in ARTMC, each node is labelled by the data stored in it and the pointer variables currently pointing to it. All program statements are encoded as (possibly non-structure preserving) tree transducers. The encoding is fully automated. The only allowed destructive pointer updates (i.e., pointer manipulating statements changing the shape of the tree) are tree rotations [6] and addition of new leaf nodes.

We have in particular considered verification of the depth-first tree traversal and the standard procedures for rebalancing red-black trees after insertion or deletion of a leaf node [6]. We have verified that the programs do not manipulate undefined and null pointers in a faulty way. For the procedures on red-black trees, we have also verified that their result is a red-black tree (without taking into account the non-regular balancedness condition). In general, the set of possible input trees for the verified procedures as well as the set of correct output trees were given as tree automata. In the case of the procedure for rebalancing red-black trees after an insertion, we have also used a generator program

Table 1. Running times (in sec.) of det. and nondet. ARTMC applied for verification of various tree manipulating programs ($\times$ denotes a too long run or a failure due to a lack of memory)

	DFT		RB-delete (null,undef)		RB-insert (null,undef)	
	det.	nondet.	det.	nondet.	det.	nondet.
full abstr.	5.2	2.7	$\times$	$\times$	33	15
restricted abstr.	40	3.5	$\times$	60	145	5.4

	RB-delete (RB preservation)		RB-insert (RB preservation)		RB-insert (gen., test.)	
	det.	nondet.	det.	nondet.	det.	nondet.
full abstr.	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
restricted abstr.	$\times$	57	$\times$	89	$\times$	978

preceding the tested procedure which generates random red-black trees and a tester program which tests the output trees being correct. Here, the set of input trees contained just an empty tree, and the verifi cation was reduced to checking that a predefi ned error location is unreachable. The size of the programs ranges from 10 to about 100 lines of pure pointer manipulations.

The results of our experiments on an Intel Xeon processor at 2.7GHz with 16GB of available memory (as in Section 3.3) are summarised in Table 1. The predicate abstraction proved to give much better results (therefore we do not consider the fi nite-height abstraction here). The abstraction was either applied after fi ring each statement of the program ("full abstraction") or just when reaching a loop point in the program ("restricted abstraction"). The results we have obtained are very encouraging and show a signifi cant improvement in the effi ciency of ARTMC based on nondeterministic tree automata. Indeed, the ARTMC framework based on deterministic tree automata has either been signifi cantly slower in the experiments (up to 25-times) or has completely failed (a too long running time or a lack of memory)—the latter case being quite frequent.

6 Conclusion

We have proposed new antichain-based algorithms for universality and inclusion checking on (nondeterministic) tree automata. The algorithms have been thoroughly tested both on randomly generated automata and on automata obtained from various verifi - cation runs performed within the abstract regular tree model checking framework. The new algorithms have been proved to be signifi cantly more effi cient than the classical determinisation-based approaches to universality and inclusion checking. Moreover, using the proposed inclusion checking algorithm together with some other recently published results, we have implemented a complete abstract regular tree model checking framework based on nondeterministic tree automata and tested it on verifi cation of several real-life pointer-intensive procedures. The results show a very encouraging improvement in the capabilities of the framework. In the future, we would like to implement the antichain-based universality and inclusion checking algorithms (as well as other recently proposed algorithms for dealing with NTA, such as the simulation-based reduction algorithms) on automata symbolically encoded as in the MONA tree automata library [9]. We hope that this will yield another signifi cant improvement in the tree automata technology allowing for a new generation of tools using tree automata (including, e.g., the abstract regular tree model checking framework).

Acknowledgement. The work was supported in part by the ANR-06-SETI-001 French project AVERISS, the Czech Grant Agency (projects 102/07/0322 and 102/05/H050), the Czech-French Barrande project 2-06-27, and the Czech Ministry of Education by the project MSM 0021630528 *Security-Oriented Research in Information Technology*.

References

1. P.A. Abdulla, A. Bouajjani, L. Hol´ık, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In *Proc. of TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.
2. P.A. Abdulla, A. Legay, J. d'Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
3. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.
4. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.
5. A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
7. L. Doyen and J.-F. Raskin. Improved Algorithms for the Automata-based Approach to Model Checking. In *Proc. of TACAS'07*, volume 4424 of *LNCS*. Springer, 2007.
8. T. Genet. Timbuk: A Tree Automata Library. http://www.irisa.fr/lande/genet/timbuk.
9. N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.
10. E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Inst. of Science, Rehovot, Israel, 2001.
11. M. De 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*, volume 4144 of *LNCS*. Springer, 2006.