

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: July 17, 2008

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

Ahmed Bouajjani¹, Peter Habermehl^{1,2}, Lukáš Holík³, Tayssir Touili¹, and Tomáš Vojnar³

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 (NTA). We have implemented these algorithms in a prototype tool and our experiments show that they provide a significant improvement over the traditional determinisation-based approaches. We use our antichain-based inclusion checking algorithm to build an abstract regular tree model checking framework based entirely on NTA. We show the significantly improved efficiency of this framework through a series of experiments with verifying various programs over dynamic linked tree-shaped data structures.

1 Introduction

Tree automata are useful in numerous different areas, including, e.g., the implementation of decision procedures for various logics, XML manipulation, linguistics or formal verification 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 are deterministic. However, as our own practical experience discussed later in the paper shows, the determinisation step may yield automata being too large to be handled although the original nondeterministic automata are quite small. It may even be the case that the corresponding minimal deterministic automata are small, but they cannot be computed as the intermediary automata resulting from determinisation are too big.

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

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 determinisation-based approaches. The experiments were done 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 automated formal verification of various kinds of infinite-state and parameterised systems. In particular, we consider its use for verification of *programs manipulating dynamic tree-shaped data structures*, and we show that the use of nondeterministic instead of deterministic tree automata improves significantly 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 (using 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 its abstract version 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 [1] on simulation-based reduction of tree automata.

2 Preliminaries

An alphabet Σ is ranked if it is endowed with a mapping $rank : \Sigma \rightarrow \mathbb{N}$. For $k \geq 0$, $\Sigma_k = \{f \in \Sigma \mid rank(f) = k\}$ is the set of symbols of rank k . The set T_Σ of terms over Σ is defined inductively: if $k \geq 0$, $f \in \Sigma_k$, and $t_1, \dots, t_k \in T_\Sigma$, then $f(t_1, \dots, t_k)$ is in T_Σ . We abbreviate the so-called *leaf terms* of the form $a()$, $a \in \Sigma_0$, by simply a . A (nondeterministic, bottom-up) *tree automaton* (NTA) is a tuple $\mathcal{A} = (Q, \Sigma, F, \delta)$ 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 $f(q_1, \dots, q_n) \rightarrow q$ where $n \geq 0$, $f \in \Sigma_n$, and $q_1, \dots, q_n, q \in Q$. We abbreviate the *leaf rules* of the form $a() \rightarrow q$, $a \in \Sigma_0$, as $a \rightarrow q$. Let t be a term over Σ . A bottom-up run of \mathcal{A} on t is obtained as follows: first, we assign a state to each leaf according to the leaf rules in δ , then for each internal node, we collect the states assigned to all its children and associate a state to the node itself according to the non-leaf δ rules. Formally, if during the state assignment process subterms t_1, \dots, t_n are labelled with states q_1, \dots, q_n , and if a rule $f(q_1, \dots, q_n) \rightarrow q$ is in δ , which we will denote by $f(q_1, \dots, q_n) \rightarrow_\delta q$, then the term $f(t_1, \dots, t_n)$ can be labelled with q . A term t is accepted if \mathcal{A} reaches its root in a final state. The language accepted by the automaton \mathcal{A} is the set of terms that it accepts: $\mathcal{L}(\mathcal{A}) = \{t \in T_\Sigma \mid t \xrightarrow{*}_\delta q(t) \text{ and } q \in F\}$.

A tree automaton is *complete* if for all $n \geq 0$, $f \in \Sigma_n$, $q_1, \dots, q_n \in Q$, there is at least one $q \in Q$ such that $f(q_1, \dots, q_n) \rightarrow_\delta q$. A tree automaton may in general

be *nondeterministic*—we call it *deterministic* if there is at most one $q \in Q$ such that $f(q_1, \dots, q_n) \rightarrow_\delta q$ for any $n \geq 0$, $f \in \Sigma_n$, $q_1, \dots, q_n \in Q$.

3 Universality Checking

Lattices and Antichains. The following definitions are similar to the corresponding ones in [11]. Let Q be a finite set. An *antichain* over Q is a set $S \subseteq 2^Q$ s.t. $\forall s, s' \in S : s \not\subseteq s'$, 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 $\forall s' \in S : s' \not\subseteq s$. Given a set $S \subseteq 2^Q$, $\lfloor S \rfloor$ denotes 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 $S \sqcup S' = \lfloor \{s \cup s' \mid s \in S \wedge s' \in S'\} \rfloor$ and the \sqsubseteq -glb (greatest lower bound) is the antichain $S \sqcap S' = \lfloor \{s \mid s \in S \vee s \in S'\} \rfloor$. We extend these definitions to lub and glb of arbitrary subsets of L in the obvious way, giving the operators \bigsqcup and \bigsqcap . Then, we get a complete lattice $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \{\emptyset\}, \emptyset)$, where $\{\emptyset\}$ is the minimal element and \emptyset the maximal one.

Upward Universality Checking Using Antichains. To check universality of a tree automaton, the standard approach is to make it complete, determinise it, complement it, and check for emptiness. As determinisation is expensive, we propose here an algorithm for checking universality without determinisation. The main idea is to try to find at least one term not accepted by the automaton. For this, we perform a kind of symbolic simulation of the automaton to cover all runs necessarily leading to non-accepting states.

In the following, q, q_1, q_2, \dots denote states of NTA, s, s_1, s_2, \dots denote sets of such states, and S, S_1, S_2, \dots denote antichains of sets of states. We assume dealing with complete automata and first give some definitions. For $f \in \Sigma_n, n \geq 0$, $Post_f^s(s_1, \dots, s_n) = \{q \mid \exists q_i \in s_i, 1 \leq i \leq n : f(q_1, \dots, q_n) \rightarrow_\delta q\}$. We omit δ if no confusion arises. Note that, for $a \in \Sigma_0$, $Post_a(\emptyset) = \{q \mid a \rightarrow_\delta q\}$ is the set of states that may be assigned to the leaf a , and $Post_f(\emptyset) = \emptyset$ for $f \in \Sigma_n, n \geq 1$. Let $Post(S) = \lfloor \{Post_f(s_1, \dots, s_n) \mid n \geq 0, s_1, \dots, s_n \in S, f \in \Sigma_n\} \rfloor$. Clearly, $Post$ is monotonic wrt. \sqsubseteq .

Let $Post_0(S) = S$ and for all $i > 0$, $Post_i(S) = Post(Post_{i-1}(S)) \sqcap S$. Intuitively, $Post_i(S)$ contains the \sqsubseteq -smallest sets $s \subseteq Q$ of states into which the automaton can nondeterministically get after processing a term of height up to i starting from the states in the elements of S . Using only the minimal sets is enough as we just need to know if there is a term on which the given automaton runs exclusively into non-final states. This makes universality checking easier than determinisation using the general subset construction.

Clearly, $Post_1(S) = Post(Post_0(S)) \sqcap S \sqsubseteq S = Post_0(S)$. Moreover, for $i > 0$, if $Post_i(S) \sqsubseteq Post_{i-1}(S)$, then due to the monotonicity of $Post$, $Post(Post_i(S)) \sqsubseteq Post(Post_{i-1}(S))$, $Post(Post_i(S)) \sqcap S \sqsubseteq Post(Post_{i-1}(S)) \sqcap S$, and therefore $Post_{i+1}(S) \sqsubseteq Post_i(S)$. Altogether, we get (1) $\forall S \in L \forall i \geq 0 : Post_{i+1}(S) \sqsubseteq Post_i(S)$. 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 $\mathcal{A} = (Q, \Sigma, F, \delta)$ be a tree automaton and t a term over Σ . Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$, then $Post^*(\emptyset) \sqsubseteq \{s\}$.*

Proof. We proceed by structural induction on t . For the *basic case*, let $t = a \in \Sigma_0$. Then, $s = \{q \mid a \rightarrow_\delta q\} = Post_a(\emptyset)$, and thus there is $s' \in Post(\emptyset)$ 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 $s'' \in Post^*(\emptyset)$. For the *induction step*, let $t = f(t_1, \dots, t_n)$. Let $s_i = \{q \in Q \mid t_i \xrightarrow{*}_\delta q\}$ for $i \in \{1, \dots, n\}$. Let $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Then, $s = \{q \mid \exists q_1 \in s_1, \dots, q_n \in s_n : f(q_1, \dots, q_n) \rightarrow_\delta q\}$. By induction, there exists $s'_i \subseteq s_i$ s.t. $s'_i \in Post^*(\emptyset)$. Let $s' = Post_f(s'_1, \dots, s'_n)$. Then, by definition of $Post_f$, we have $s' \subseteq s$, and by definition of $Post^*$, there exists $s'' \subseteq s'$ with $s'' \in Post^*(\emptyset)$. \square

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

Proof. Let $i \geq 1$ be the smallest index s.t. $s \in Post_i(\emptyset)$. We proceed by induction on i . For the *basic case*, $i = 1$. Then, there is $a \in \Sigma_0$ s.t. $s = Post_a(\emptyset) = \{q \mid a \xrightarrow{*}_\delta q\}$, $t = a$. For the *induction step*, let $i > 1$. There exists $f \in \Sigma_n$ and $s_1, \dots, s_n \in Post_{i-1}(\emptyset)$ with $s = Post_f(s_1, \dots, s_n)$. By induction, there exists t_1, \dots, t_n s.t. for $j \in \{1, \dots, n\}$, $s_j = \{q \mid t_j \xrightarrow{*}_\delta q\}$. Let $t = f(t_1, \dots, t_n)$. By definition of $Post_f$, $s = \{q \mid t \xrightarrow{*}_\delta q\}$. \square

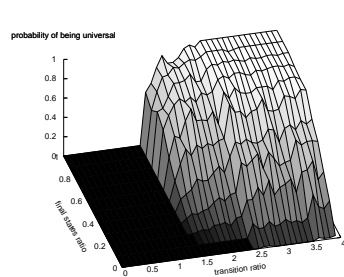
We can now give a theorem allowing us to decide universality *without determinisation*.

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

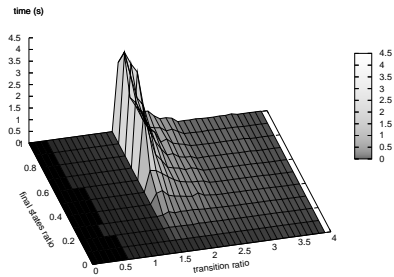
Proof. Let \mathcal{A} be not universal. Let t be a term not accepted by \mathcal{A} and $s = \{q \mid t \xrightarrow{*}_\delta q\}$. As t is not accepted by the automaton, $s \subseteq \overline{F}$. By Lemma 1, there is $s' \in Post^*(\emptyset)$ s.t. $s' \subseteq s \subseteq \overline{F}$. Suppose now that there exists $s \in Post^*(\emptyset)$ s.t. $s \subseteq \overline{F}$. By Lemma 2, there exists a term t with $s = \{q \mid t \xrightarrow{*}_\delta q\}$. Since $s \subseteq \overline{F}$, t is not accepted by \mathcal{A} . \square

Experiments with Upward Universality Checking Using Antichains. We have implemented the above approach for testing universality of tree automata in a prototype based on the Timbuk tree automata library [8]. We give the results of our experiments run on an Intel Xeon processor at with 2.7GHz and 16GB of memory 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.

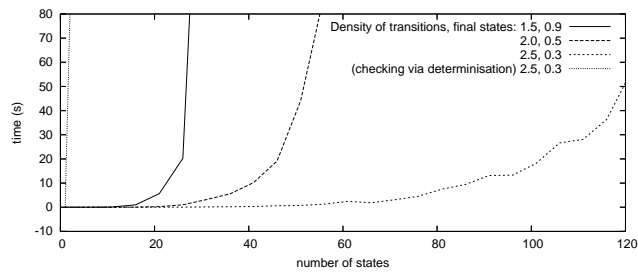
In the random tests, we first used automata with 20 states and varied the *density of their transitions* (the average number of different right-hand side states for a given left-hand side of a transition rule, i.e., $|\delta|/|\{f(q_1, \dots, q_n) \mid \exists q \in Q : f(q_1, \dots, q_n) \rightarrow_\delta q\}|$) 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 their universality using our antichain-based approach. The 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 chosen 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. We also show the time needed when 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 also be drawn from the results of Fig. 1(d) obtained on automata from experimenting with abstract regular tree model checking applied for verifying various procedures manipulating trees presented in Section 5.3.



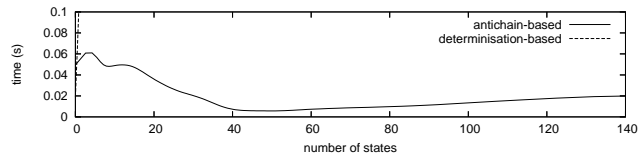
(a) Probability that a tree automaton (TA) with 20 states and some density of transitions and final states is universal



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



(c) Universality checking via determinisation and antichains 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

Downward Universality Checking with Antichains. The upward universality checking introduced above for tree automata conceptually corresponds to the forward universality checking of finite word automata of [11]. In [11], a dual backward universality checking is also introduced. It is based on computing the *controllable predecessors* of the set of non-final states. Controllable predecessors are the predecessors that can be forced by an

input symbol to continue into a given set of states. Then, the automaton is non-universal iff the controllable predecessors of the non-final states cover the set of initial states.

Downward universality checking for tree automata as a dual approach to upward universality checking is problematic since the controllable predecessors of a set of states $s \subseteq Q$ of an NTA $\mathcal{A} = (Q, \Sigma, F, \delta)$ do not form a set of states, but a set of *tuples* of states, i.e., $CPre(s) = \{(q_1, \dots, q_n) \mid n \in \mathbb{N} \wedge \exists f \in \Sigma \forall q \in Q : f(q_1, \dots, q_n) \xrightarrow{\delta} q \Rightarrow q \in s\}$. Note that if we flatten the set $CPre(s)$ to the set $FCCPre(s)$ of states that appear in some of the tuples of $CPre(s)$ and check that starting from leaf rules the computation can be forced into some subset of $FCCPre(s)$, then this does not imply that the computation can be forced into some state from s . That is because for any rule $f(q_1, \dots, q_n) \xrightarrow{\delta} q, q \in s$, not all of the states q_1, \dots, q_n may be reached. Moreover, it is too strong to require that starting from leaf rules, it must be possible to force the computation into all states of $FCCPre_f(s)$. Clearly, it is enough if the computation starting from leaf rules can be forced into s via some of the vectors in $CPre(s)$, not necessarily all of them. Also, if we keep $CPre(s)$ for $s \subseteq Q$ as a set of vectors, we also have to define the notion of controllable predecessors for sets of vectors of states, which is a set of vectors of vectors of states, etc. Clearly, such an approach is not practical.

4 Inclusion Checking

Let $\mathcal{A} = (Q, \Sigma, F, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', \delta')$ be two tree automata. We want to check if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. The traditional approach computes the complement of \mathcal{B} and checks if it has an empty intersection with \mathcal{A} . This is costly as computing the complement necessitates determinisation. Here we show how to check inclusion without determinisation.

As before, the idea is to find at least one term accepted by \mathcal{A} and not by \mathcal{B} . For that, we simultaneously simulate the runs of the two automata using pairs (p, s) with $p \in Q$ and $s \subseteq Q'$ where p memorises the run of \mathcal{A} and s all the possible runs of \mathcal{B} . If t is a term accepted by \mathcal{A} and not by \mathcal{B} , the simultaneous run of the two automata on t 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 to make sure that no run of \mathcal{B} can accept the term t . Therefore, s must 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 $s \not\subseteq s'$. We denote by L_I the set of all antichains over $Q \times 2^{Q'}$. Given a set $\mathcal{S} \subseteq Q \times 2^{Q'}$, an element $(p, s) \in \mathcal{S}$ is *minimal* if for every $s' \subset s$, $(p, s') \notin \mathcal{S}$. We denote by $|\mathcal{S}|$ the set of minimal elements of \mathcal{S} . Given two antichains \mathcal{S} and \mathcal{S}' , we define 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 is $(p, s) \in \mathcal{S}$ s.t. $s \subseteq s'$; $\mathcal{S} \sqcup_I \mathcal{S}' = \lfloor \{(p, s \cup s') \mid (p, s) \in \mathcal{S} \wedge (p, s') \in \mathcal{S}'\} \rfloor$; and $\mathcal{S} \sqcap_I \mathcal{S}' = \lfloor \{(p, s) \mid (p, s) \in \mathcal{S} \vee (p, s) \in \mathcal{S}'\} \rfloor$. These definitions can be extended to arbitrary sets in the usual way leading to the operators \sqcup_I and \sqcap_I , yielding a complete lattice as in Section 3.

Given $f \in \Sigma_n, n \geq 0$, we define $IPost_f((p_1, s_1), \dots, (p_n, s_n)) = \{(p, s) \mid f(p_1, \dots, p_n) \xrightarrow{\delta} p \wedge s = Post_f^{\delta'}(s_1, \dots, s_n)\}$. Let \mathcal{S} be an antichain over $Q \times 2^{Q'}$. Then, let $IPost(\mathcal{S}) = \lfloor \{IPost_f((p_1, s_1), \dots, (p_n, s_n)) \mid n \geq 0, (p_1, s_1), \dots, (p_n, s_n) \in \mathcal{S}, f \in \Sigma_n\} \rfloor$. Let $IPost_0(\mathcal{S}) = \mathcal{S}$ and $IPost_i(\mathcal{S}) = IPost(IPost_{i-1}(\mathcal{S})) \sqcap_I \mathcal{S}$. As before, we can show that $\forall \mathcal{S} \in L_I \forall i \geq 0 : IPost_{i+1}(\mathcal{S}) \sqsubseteq_I IPost_i(\mathcal{S})$, and

that for every antichain \mathcal{S} , there exists a J such that $I\text{Post}_{J+1}(\mathcal{S}) = I\text{Post}_J(\mathcal{S})$. Let $I\text{Post}^*(\mathcal{S}) = I\text{Post}_J(\mathcal{S})$. Note that, like in the case of $\text{Post}_a(\emptyset)$ in Section 3, $I\text{Post}_a(\emptyset) = \{(q, \text{Post}_a^{\delta'}(\emptyset)) \mid a \rightarrow_\delta q\}$ for $a \in \Sigma_0$, and $I\text{Post}_f(\emptyset) = \emptyset$ for $f \in \Sigma_n$, $n \geq 1$. Then, we get the following lemma. The proof is similar to the one of Lemma 1.

Lemma 3. *Let $\mathcal{A} = (Q, \Sigma, F, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', \delta')$ be two tree 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, $I\text{Post}^*(\emptyset) \sqsubseteq_I \{(p, s)\}$.*

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

Lemma 4. *Let $\mathcal{A} = (Q, \Sigma, F, \delta)$ and $\mathcal{B} = (Q', \Sigma, F', \delta')$ be two tree automata, and let $(p, s) \in I\text{Post}^*(\emptyset)$. Then there is a term t over Σ s.t. $t \xrightarrow{*}_\delta p$ and $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$.*

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

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

Proof. Suppose that $(p, s) \in I\text{Post}^*(\emptyset)$ with $p \in F$ and $s \subseteq \overline{F'}$. Using Lemma 4 there is a term t with $t \xrightarrow{*}_\delta p$ and $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$. As $p \in F$ 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 $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$. Let 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 $s = \{q \mid t \xrightarrow{*}_{\delta'} q\}$. Then, $s \subseteq \overline{F'}$. Lemma 3 implies that $I\text{Post}^*(\emptyset)$ contains a pair (p, s') s.t. $s' \subseteq s \subseteq \overline{F'}$. \square

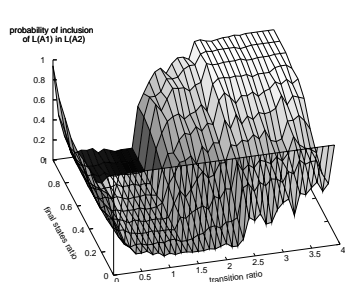
4.1 Experiments with Inclusion Checking Using Antichains

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).

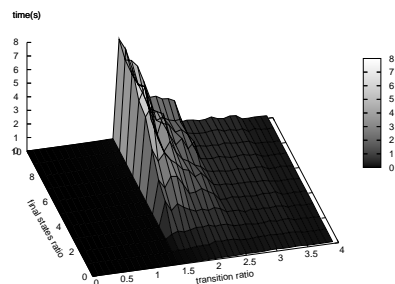
We first ran our tests on pairs of randomly generated automata having 10 states and different possible densities of transitions and final states. The probability that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ holds for randomly generated tree automata \mathcal{A}_1 and \mathcal{A}_2 (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 .

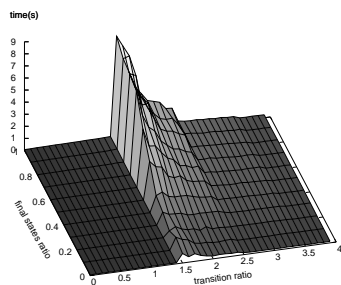
Finally, in Fig. 3(a), we show how the running times change for some selected instances of the problem (in terms of some selected densities of transitions and final



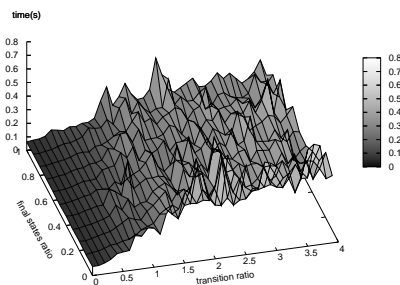
(a) Probability of $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ for tree automata (TA) with 10 states and some density of transitions and final states



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



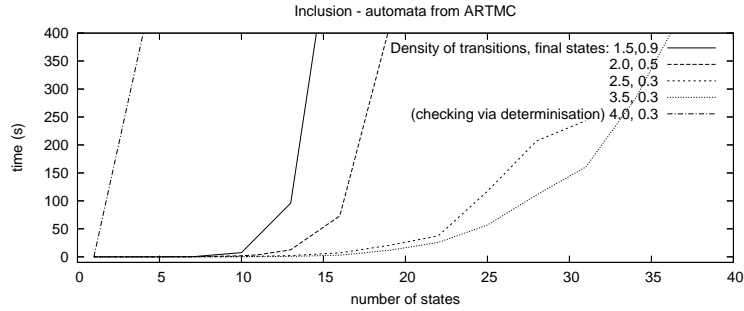
(c) Antichain-based inclusion checking on TA, \mathcal{A}_1 random, \mathcal{A}_2 with some density of transitions and final states



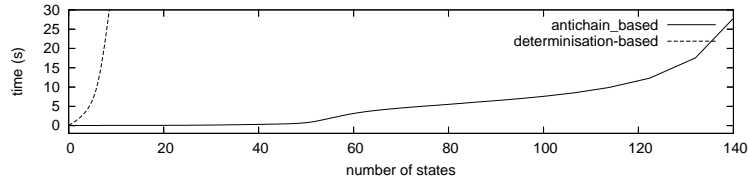
(d) Antichain-based inclusion checking on TA, \mathcal{A}_2 random, \mathcal{A}_1 with some density of transitions and final states

Fig. 2. Experiments with inclusion checking on tree automata

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 $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. 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.



(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 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 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.

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}_Σ the set of all tree automata over Σ . Let $\mathcal{I} \in \mathbb{M}_\Sigma$

be a tree automaton describing a set of initial configurations, τ a term rewriting system describing the behaviour of a system, and $\mathcal{B} \in \mathbb{M}_{\mathcal{S}}$ a tree automaton describing a set of bad configurations. The safety verification problem can now be formulated as checking whether the following holds:

$$\tau^*(\mathcal{L}(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) = \emptyset \quad (1)$$

In ARTMC, the precise set of reachable configurations $\tau^*(\mathcal{L}(\mathcal{I}))$ is not computed to solve Problem (1). Instead, its overapproximation is computed by interleaving the application of τ and the union in $\mathcal{L}(\mathcal{I}) \cup \tau(\mathcal{L}(\mathcal{I})) \cup \tau(\tau(\mathcal{L}(\mathcal{I}))) \cup \dots$ with an application of an abstraction function α . The abstraction is applied on the tree automata encoding the so-far computed sets of reachable configurations.

An abstraction function is defined as a mapping $\alpha : \mathbb{M}_{\mathcal{S}} \rightarrow \mathbb{A}_{\mathcal{S}}$ where $\mathbb{A}_{\mathcal{S}} \subseteq \mathbb{M}_{\mathcal{S}}$ and $\forall \mathcal{A} \in \mathbb{M}_{\mathcal{S}} : \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall \mathcal{A} \in \mathbb{M}_{\mathcal{S}} : \mathcal{L}(\alpha'(\mathcal{A})) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. Given a term rewriting system τ and an abstraction α , a mapping $\tau_{\alpha} : \mathbb{M}_{\mathcal{S}} \rightarrow \mathbb{M}_{\mathcal{S}}$ is defined as $\forall \mathcal{A} \in \mathbb{M}_{\mathcal{S}} : \tau_{\alpha}(\mathcal{A}) = \hat{\tau}(\alpha(\mathcal{A}))$ 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 $\mathbb{A}_{\mathcal{S}}$ 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 (1) 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:

$$\mathcal{I}, \tau_{\alpha}(\mathcal{I}), \tau_{\alpha}^2(\mathcal{I}), \dots, \tau_{\alpha}^{n-1}(\mathcal{I}), \tau_{\alpha}^n(\mathcal{I}) \quad (2)$$

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 (1) 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 $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$

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

Table 1. Running times (in sec.) of det. and nondet. ARTMC applied for verification of various tree manipulating programs (× 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	×	×	33	15
restricted abstr.	40	3.5	×	60	145	5.4
	RB-delete (RB preservation)		RB-insert (RB preservation)		RB-insert (gen., test.)	
	det.	nondet.	det.	nondet.	det.	nondet.
full abstr.	×	×	×	×	×	×
restricted abstr.	×	57	×	89	×	978

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 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 verification was reduced to checking that a predefined 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) are summarised in Table 1. The predicate abstraction proved to give much better results (therefore we do not consider the finite-height abstraction here). The abstraction was either applied after firing 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 significant improvement in the efficiency of ARTMC based on nondeterministic tree automata. Indeed, the ARTMC framework based on deterministic tree automata has either been significantly 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 verification runs performed within the abstract regular tree model checking framework. The new algorithms have been proved to be significantly more efficient 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 verification 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 significant 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. Holík, 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. Rezzina. 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.