Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata[⋆]

Lukáš Holík[1,2], Ondřej Lengál[1], Jiří Šimáček[1,3], and Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, Czech Republic
[2] Uppsala University, Sweden
[3] VERIMAG, UJF/CNRS/INPG, Gières, France

Abstract. The paper considers several issues related to efficient use of tree automata in formal verification. First, a new efficient algorithm for inclusion checking on non-deterministic tree automata is proposed. The algorithm traverses the automaton downward, utilizing antichains and simulations to optimize its run. Results of a set of experiments are provided, showing that such an approach often very significantly outperforms the so far common upward inclusion checking. Next, a new semi-symbolic representation of non-deterministic tree automata, suitable for automata with huge alphabets, is proposed together with algorithms for upward as well as downward inclusion checking over this representation of tree automata. Results of a set of experiments comparing the performance of these algorithms are provided, again showing that the newly proposed downward inclusion is very often better than upward inclusion checking.

1 Introduction

Finite tree automata play a crucial role in several formal verification techniques, such as (abstract) regular tree model checking [3, 5], verification of programs with complex dynamic data structures [6, 11], analysis of network firewalls [7], and implementation of decision procedures of logics such as WS2S or MSO [15], which themselves have numerous applications (among the most recent and promising ones, let us mention at least verification of programs manipulating heap structures with data [16]).

Recently, there has been notable progress in the development of algorithms for efficient manipulation of non-deterministic finite tree automata (TA), more specifically, in solving the crucial problems of automata reduction [1] and of checking language inclusion [18, 4, 2]. As shown, e.g., in [4], replacing deterministic automata by non-deterministic ones can—in combination with the new methods for handling TA—lead to great efficiency gains. In this paper, we further advance the research on efficient algorithms for handling TA by (i) proposing a new algorithm for inclusion checking that turns out to significantly outperform the existing algorithms in most of our experiments and (ii) by presenting a semi-symbolic multi-terminal binary decision diagram (MTBDD) based representation of TA, together with various important algorithms for handling TA working over this representation.

The classic textbook algorithm for checking inclusion $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ between two TA $\mathcal{A}_S$ (Small) and $\mathcal{A}_B$ (Big) first determinizes $\mathcal{A}_B$, computes the complement automaton $\overline{\mathcal{A}_B}$ of $\mathcal{A}_B$, and then checks language emptiness of the product automaton accepting

⋆ This work was supported by the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the BUT FIT project FIT-S-11-1, and the Swedish UPMARC project.

$\mathcal{L}(\mathcal{A}_S) \cap \mathcal{L}(\overline{\mathcal{A}_B})$. This approach has been optimized in [18, 4, 2] which describe variants of this algorithm that try to avoid constructing the whole product automaton (which can be exponentially larger than $\mathcal{A}_B$ and which is indeed extremely large in many practical cases) by constructing its states and checking language emptiness on the fly. By employing the antichain principle [18, 4], possibly combined with using upward simulation relations [2], the algorithm is often able to prove or refute inclusion by constructing a small part of the product automaton only.[4] We denote these algorithms as *upward* algorithms to reflect the direction in which they traverse automata $\mathcal{A}_S$ and $\mathcal{A}_B$.

The upward algorithms are sufficiently efficient in many practical cases. However, they have two drawbacks: (i) When generating the bottom-up post-image of a set $\mathcal{S}$ of sets of states, all possible n-tuples of states from all possible products $S_1 \times \ldots \times S_n$, $S_i \in \mathcal{S}$ need to be enumerated. (ii) Moreover, these algorithms are known to be compatible with only upward simulations as a means of their possible optimization, which is a disadvantage since downward simulations are often richer and also cheaper to compute.

The alternative *downward* approach to checking TA language inclusion was first proposed in [13] in the context of subtyping of XML types. This algorithm is not derivable from the textbook approach and has a more complex structure with its own weak points; nevertheless, it does not suffer from the two issues of the upward algorithm mentioned above. We generalize the algorithm of [13] for automata over alphabets with an arbitrary rank ([13] considers rank at most two), and, most importantly, we improve it significantly by using the antichain principle, empowered by a use of the cheap and usually large downward simulation. In this way, we obtain an algorithm which is complementary to and highly competitive with the upward algorithm as shown by our experimental results (in which the newly proposed algorithm significantly dominates in most of the considered cases).

Certain important applications of TA such as formal verification of programs with complex dynamic data structures or decision procedures of logics such as WS2S or MSO require handling very large alphabets. Here, the common choice is to use the MONA tree automata library [15] which is based on representing transitions of TA symbolically using MTBDDs. However, the encoding used by MONA is restricted to *deterministic* automata only. This implies a necessity of immediate determinisation after each operation over TA that introduces nondeterminism, which very easily leads to a state space explosion. Despite the extensive engineering effort spent to optimize the implementation of MONA, this fact significantly limits its applicability.

As a way to overcome this difficulty, we propose a semi-symbolic representation of *non-deterministic* TA which generalises the one used by MONA, and we develop algorithms implementing the basic operations on TA (such as union, intersection, etc.) as well as more involved algorithms for computing simulations and for checking inclusion (using simulations and antichains to optimize it) over the proposed representation. We also report on experiments with a prototype implementation of our algorithms showing

[4] The work of [18] does, in fact, not use the terminology of antichains despite implementing them in a symbolic, BDD-based way. It specialises to binary tree automata only. A more general introduction of antichains within a lattice-theoretic framework appeared in the context of word automata in [19]. Subsequently, [4] has generalized [19] for explicit upward inclusion checking on TA and experimentally advocated its use within abstract regular tree model checking [4]. See also [10] for other combinations of antichains and simulations for word automata.

again a dominance of downward inclusion checking and justifying usefulness of our symbolic encoding for TA with large alphabets.

The rest of this paper is organised as follows. Section 2 contains basic definitions for tree automata, tree automata languages, and simulations. Section 3 describes our downward inclusion checking algorithm and its experimental comparison with the upward algorithms. Further, Section 4 presents our MTBDD-based TA encoding, the algorithms working over this encoding, and an experimental evaluation of these algorithms. Section 5 then concludes the paper.

2 Preliminaries

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_Σ the set of all trees over the alphabet Σ.

A (finite, non-deterministic) *tree automaton* (abbreviated sometimes as TA in the following) 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 equivalently $(q_1, \ldots, q_n) \xrightarrow{a} q$ and $q \xrightarrow{a} (q_1, \ldots, q_n)$ to denote that $((q_1, \ldots, q_n), a, q) \in \Delta$. The two notations correspond to the bottom-up and top-down representation of tree automata, respectively. (Note that we can afford to work interchangeably with both of them since we work with non-deterministic tree automata, which are known to have an equal expressive power in their bottom-up and top-down representations.) In the special case when $n = 0$, we speak about the so-called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{a} q$ or $q \xrightarrow{a}$.

For an automaton $\mathcal{A} = (Q, \Sigma, \Delta, F)$, we use $Q^\#$ to denote the set of all tuples of states from Q with up to the maximum arity that some symbol in Σ has, i.e., if $r = max_{a \in \Sigma} \#a$, then $Q^\# = \bigcup_{0 \leq i \leq r} Q^i$. For $p \in Q$ and $a \in \Sigma$, we use $down_a(p)$ to denote the set of tuples accessible from p over a in the top-down manner; formally, $down_a(p) = \{(p_1, \ldots, p_n) \mid p \xrightarrow{a} (p_1, \ldots, p_n)\}$. For $a \in \Sigma$ and $(p_1, \ldots, p_n) \in Q^{\#a}$, we denote by $up_a((p_1, \ldots, p_n))$ the set of states accessible from $(p_1, \ldots, p_n)$ over the symbol a in the bottom-up manner; formally, $up_a((p_1, \ldots, p_n)) = \{p \mid (p_1, \ldots, p_n) \xrightarrow{a} p\}$. We also extend these notions to sets in the usual way, i.e., for $a \in \Sigma$, $P \subseteq Q$, and $R \subseteq Q^{\#a}$, $down_a(P) = \bigcup_{p \in P} down_a(p)$ and $up_a(R) = \bigcup_{(p_1, \ldots, p_n) \in R} up_a((p_1, \ldots, p_n))$.

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 rank $\#t(v) = n$ 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 $t \overset{\pi}{\Longrightarrow} q$ 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 $t \overset{\pi}{\Longrightarrow} q$ 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 a set of states $S \subseteq Q$ is defined as $\mathcal{L}_{\mathcal{A}}(S) = \bigcup_{q \in S} \mathcal{L}_{\mathcal{A}}(q)$. When it is clear which TA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathcal{A}}(F)$. We also extend the notion of a language to a tuple of states $(q_1,\ldots,q_n) \in Q^n$ by letting $\mathcal{L}((q_1,\ldots,q_n)) = \mathcal{L}(q_1) \times \cdots \times \mathcal{L}(q_n)$. The language of a set of n-tuples of sets of states $S \subseteq (2^Q)^n$ is the union of languages of elements of S, the set $\mathcal{L}(S) = \bigcup_{E \in S} \mathcal{L}(E)$. We say that X accepts y to express that $y \in \mathcal{L}(X)$.

A *downward simulation* on TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ is a preorder relation $\preceq_D \subseteq Q \times Q$ such that if $q \preceq_D p$ and $(q_1,\ldots,q_n) \xrightarrow{a} q$, then there are states $p_1,\ldots,p_n$ such that $(p_1,\ldots,p_n) \xrightarrow{a} p$ and $q_i \preceq_D p_i$ for each $1 \le i \le n$. Given a TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ and a downward simulation $\preceq_D$, an *upward simulation* $\preceq_U \subseteq Q \times Q$ induced by $\preceq_D$ is a relation such that if $q \preceq_U p$ and $(q_1,\ldots,q_n) \xrightarrow{a} q'$ with $q_i = q$, $1 \le i \le n$, then there are states $p_1,\ldots,p_n,p'$ such that $(p_1,\ldots,p_n) \xrightarrow{a} p'$ where $p_i = p$, $q' \preceq_U p'$, and $q_j \preceq_D p_j$ for each j such that $1 \le j \ne i \le n$.

3 Downward Inclusion Checking

Let us fix two tree automata $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S)$ and $\mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B)$ for which we want to check whether $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ holds. If we try to answer this query top-down and we proceed in a naïve way, we immediately realize that the fact that the top-down successors of particular states are *tuples* of states leads us to checking inclusion of the languages of tuples of states. Subsequently, the need to compare the languages of each corresponding pair of states in these tuples will again lead to comparing the languages of tuples of states, and hence, we end up comparing the languages of *tuples of tuples* of states, and the need to deal with more and more nested tuples of states never stops.

For instance, given a transition $q \xrightarrow{a} (p_1,p_2)$ in $\mathcal{A}_S$, transitions $r \xrightarrow{a} (s_1,s_2)$ and $r \xrightarrow{a} (t_1,t_2)$ in $\mathcal{A}_B$, and assuming that there are no further top-down transitions from q and r, it holds that $\mathcal{L}(q) \subseteq \mathcal{L}(r)$ if and only if $\mathcal{L}((p_1,p_2)) \subseteq \mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2))$. Note that the union $\mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2))$ cannot be computed component-wise, this is, $\mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2)) \ne (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2))$. For instance, provided $\mathcal{L}(s_1) = \mathcal{L}(s_2) = \{b\}$ and $\mathcal{L}(t_1) = \mathcal{L}(t_2) = \{c\}$, it holds that $\mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2)) = \{(b,b),(c,c)\}$, but the component-wise union is $(\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2)) = \{(b,b),(b,c),(c,b),(c,c)\}$. Hence, we cannot simply check whether $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$ and $\mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2)$ to answer the original query, and we have to proceed by checking inclusion on the obtained tuples of states. However, exploring the top-down transitions that lead from the states that appear in these tuples will lead us to dealing with tuples of tuples of states, etc.

Fortunately, there is a way out of the above trap. In particular, as first observed in [13] in the context of XML type checking, we can exploit the following property of the Cartesian product of sets $G,H \subseteq \mathcal{U}$: $G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$.

Hence, when we continue with our example, we get $\mathcal{L}((p_1,p_2)) = \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq \mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2)) = (\mathcal{L}(s_1) \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times \mathcal{L}(t_2)) = ((\mathcal{L}(s_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(s_2))) \cup ((\mathcal{L}(t_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(t_2)))$. This can further be rewritten, using the distributive laws in the $(2^{T_\Sigma \times T_\Sigma}, \subseteq)$ lattice, as $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \cap ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2)))$. It is easy to see that the inclusion holds exactly if it holds for all components of the intersection, i.e., if and only if $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup$

$(\mathcal{L}(t_1) \times T_\Sigma)) \;\wedge\; \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \;\wedge\; \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \;\wedge\; \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2)))$.

Two things should be noted in the above condition: (1) If we are computing the union of languages of two tuples such that they have T_Σ at all indices other than some index i, we can compute it component-wise. For instance, $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) = (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times T_\Sigma$. This clearly holds iff $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$. (2) If T_Σ does not appear at the same positions as in the inclusion $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2)))$, it must hold that either $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1)$ or $\mathcal{L}(p_2) \subseteq \mathcal{L}(t_2)$.

Using the above observations, we can finally rewrite the equation $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq \mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2))$ into the following formula that does not contain languages of tuples but of single states only: $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1) \;\wedge\; (\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(t_2)) \;\wedge\; (\mathcal{L}(p_1) \subseteq \mathcal{L}(t_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2)) \;\wedge\; \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2)$.

The above reasoning can be generalized to dealing with transitions of any arity as shown in Theorem 1, proved in [12]. In the theorem, we conveniently exploit the notion of *choice functions*. Given $P_B \subseteq Q_B$ and $a \in \Sigma$, $\#a = n \geq 1$, we denote by $cf(P_B,a)$ the set of all choice functions f that assign an index i, $1 \leq i \leq n$, to all n-tuples $(q_1,\ldots,q_n) \in Q_B^n$ such that there exists a state in P_B that can make a transition over a to $(q_1,\ldots,q_n)$; formally, $cf(P_B,a) = \{f : down_a(P_B) \to \{1,\ldots,\#a\}\}$.

Theorem 1. *Let $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S)$ and $\mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B)$ be tree automata. For sets $P_S \subseteq Q_S$ and $P_B \subseteq Q_B$ it holds that $\mathcal{L}(P_S) \subseteq \mathcal{L}(P_B)$ if and only if $\forall p_S \in P_S\ \forall a \in \Sigma$: if $p_S \xrightarrow{a} (r_1,\ldots,r_{\#a})$,*

$$\text{then}\quad \begin{cases} down_a(P_B) = \{()\} & \text{if } \#a = 0,\\[1ex] \forall f \in cf(P_B,a)\ \exists 1 \leq i \leq \#a : \mathcal{L}(r_i) \subseteq \displaystyle\bigcup_{\substack{\overline{u} \in down_a(P_B)\\ f(\overline{u}) = i}} \mathcal{L}(u_i) & \text{if } \#a > 0. \end{cases}$$

3.1 Basic Algorithm of Downward Inclusion Checking

Next, we construct a basic algorithm for downward inclusion checking on tree automata $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S)$ and $\mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B)$. The algorithm is shown as Algorithm 1. Its main idea relies on a recursive application of Theorem 1 in function `expand1`. The function is given a pair $(p_S,P_B) \in Q_S \times 2^{Q_B}$ for which we want to prove that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$—initially, the function is called for every pair (q_S,F_B) where $q_S \in F_S$. The function enumerates all possible top-down transitions that $\mathcal{A}_S$ can do from p_S (lines 3–8). For each such transition, the function either checks whether there is some transition $p_B \xrightarrow{a}$ for $p_B \in P_B$ if $\#a = 0$ (line 5), or it starts enumerating and recursively checking queries $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ on which the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ depends according to Theorem 1 (lines 9–16).

The `expand1` function keeps track of which inclusion queries are currently being evaluated in the set *workset* (line 2). Encountering a query $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ with $(p'_S,P'_B) \in workset$ means that the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ depends on the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ itself. In this case, the function immediately successfully returns because the result of the query then depends only on the other branches of the call tree.

Using Theorem 1 and noting that Algorithm 1 necessarily terminates because all its loops are bounded, and the recursion in function `expand1` is also bounded due to the use of *workset*, it is not difficult to see that the following theorem holds.

```
Algorithm 1: Downward inclusion
  Input: Tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$
  Output: true if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, false otherwise
1 foreach $q_S \in F_S$ do
2     if $\neg$expand1($q_S, F_B, \emptyset$) then return false;
3 return true;
```

```
Function expand1 ($p_S$, $P_B$, $workset$)
   /* $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$ */
1  if $(p_S, P_B) \in workset$ then return true;
2  $workset := workset \cup \{(p_S, P_B)\}$;
3  foreach $a \in \Sigma$ do
4      if $\#a = 0$ then
5          if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ then return false;
6      else
7          $W := down_a(P_B)$;
8          foreach $(r_1, \ldots, r_{\#a}) \in down_a(p_S)$ do        /* $p_S \xrightarrow{a} (r_1, \ldots, r_{\#a})$ */
9              foreach $f \in \{W \to \{1, \ldots, \#a\}\}$ do
10                 $found := false$;
11                 foreach $1 \leq i \leq \#a$ do
12                     $S := \{q_i \mid (q_1, \ldots, q_{\#a}) \in W, f((q_1, \ldots, q_{\#a})) = i\}$;
13                     if expand1($r_i, S, workset$) then
14                         $found := true$;
15                         break;
16                 if $\neg found$ then return false;
17 return true;
```

Theorem 2. *When applied on TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$, Algorithm 1 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

3.2 Optimized Algorithm of Downward Inclusion Checking

In this section, we propose several optimizations of the basic algorithm presented above that, according to our experiments, often have a huge impact on the efficiency of the algorithm—making it in many cases the most efficient algorithm for checking inclusion on tree automata that we are currently aware of. In general, the optimizations are based on an original use of simulations and antichains in a way suitable for the context of downward inclusion checking.

In what follows, we assume that there is available a preorder $\preceq \subseteq (Q_S \cup Q_B)^2$ compatible with language inclusion, i.e., such that $p \preceq q \implies \mathcal{L}(p) \subseteq \mathcal{L}(q)$, and we use $P \preceq^{\forall\exists} R$ where $P, R \subseteq (Q_S \cup Q_B)^2$ to denote that $\forall p \in P \exists r \in R : p \preceq r$. An example of such a preorder, which can be efficiently computed, is the (maximal) downward simulation $\preceq_D$. We propose the following concrete optimizations of the downward checking of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$:

1. If there exists a state $p_B \in P_B$ such that $p_S \preceq p_B$, then the inclusion clearly holds (from the assumption made about $\preceq$), and no further checking is needed.
2. Next, it can be seen without any further computation that the inclusion does *not* hold if there exists some (p'_S, P'_B) such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists} P'_B$, and we have already established that $\mathcal{L}(p'_S) \not\subseteq \mathcal{L}(P'_B)$. Indeed, we have $\mathcal{L}(P_B) \subseteq \mathcal{L}(P'_B) \not\supseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(p_S)$, and therefore $\mathcal{L}(p_S) \not\subseteq \mathcal{L}(P_B)$.

Algorithm 2: Downward inclusion (antichains + preorder)

```
    Input: TA $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S), \mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B), \preceq \subseteq (Q_S \cup Q_B)^2$
    Output: $true$ if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, $false$ otherwise
    Data: $NN := \emptyset$
 1  foreach $q_S \in F_S$ do
 2      if $\neg$expand2$(q_S, F_B, \emptyset)$ then return $false$;
 3  return $true$;
```

Function expand2 (p_S, P_B, $workset$)

```
    /* $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$ */
 1  if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then return $true$;
 2  if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ then return $false$ ;
 3  if $\exists p \in P_B : p_S \preceq p$ then return $true$;
 4  $workset := workset \cup \{(p_S, P_B)\}$;
 5  foreach $a \in \Sigma$ do
 6      if $\#a = 0$ then
 7          if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ then return $false$;
 8      else
 9          $W := down_a(P_B)$;
10          foreach $(r_1,\ldots,r_{\#a}) \in down_a(p_S)$ do          /* $p_S \xrightarrow{a} (r_1,\ldots,r_{\#a})$ */
11              foreach $f \in \{W \to \{1,\ldots,\#a\}\}$ do
12                  $found := false$;
13                  foreach $1 \leq i \leq \#a$ do
14                      $S := \{q_i \mid (q_1,\ldots,q_{\#a}) \in W, f((q_1,\ldots,q_{\#a})) = i\}$;
15                      if expand2$(r_i, S, workset)$ then
16                          $found := true$;
17                          break;
18                      if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ then
19                          $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
20                  if $\neg found$ then return $false$;
21  return $true$;
```

3. Finally, we can stop evaluating the given inclusion query if there is some $(p'_S, P'_B) \in$ *workset* such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$. Indeed, this means that the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ depends on the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$. However, if $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ holds, then also $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ holds because we have $\mathcal{L}(p_S) \subseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B) \subseteq \mathcal{L}(P_B)$.

The version of Algorithm 1 including all the above proposed optimizations is shown as Algorithm 2. The optimizations can be found in the function `expand2` that replaces the function `expand1`. In particular, line 3 implements the first optimization, line 2 the second one, and line 1 the third one. In order to implement the second optimization, the algorithm maintains a new set *NN*. This set stores pairs (p_S, P_B) for which it has already been shown that the inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ does *not* hold.[5]

As a further optimization, the set *NN* is maintained as an antichain w.r.t. the preorder that compares the pairs stored in *NN* such that the states from Q_S on the left are compared w.r.t. $\preceq$, and the sets from 2^{Q_B} on the right are compared w.r.t. $\succeq^{\exists\forall}$ (line 19). Clearly, there is no need to store a pair (p_S, P_B) that is bigger in the described sense

[5] In [12], a further optimization exploiting that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ has been shown to hold is proposed, but it is much more complicated in order to avoid memorizing possibly invalid assumptions made during the computation.

Table 1. Percentages of cases in which the respective methods were the fastest

Size	Pairs	Timeout	`Up`	`Up+s`	`Down`	`Down+s`	Avg up speedup	Avg down speedup
50–250	323	20 s	31.21 %	0.00 %	53.50 %	15.29 %	1.71	3.55
400–600	64	60 s	9.38 %	0.00 %	39.06 %	51.56 %	0.34	46.56

than some other pair (p'_S, P'_B) since every time (p_S, P_B) can be used to prune the search, (p'_S, P'_B) can also be used.

Taking into account Theorem 2 and the above presented facts, it is not difficult to see that the following holds.

Theorem 3. *When applied on TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$, Algorithm 2 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

3.3 Experimental Results

We have implemented Algorithm 1 (which we mark as `Down` in what follows) as well as Algorithm 2 using the maximum downward simulation as the input preorder (which is marked as `Down+s` below). We have also implemented the algorithm of upward inclusion checking using antichains from [4] and its modification using upward simulation proposed in [2] (these algorithms are marked as `Up` and `Up+s` below). We tested our approach on 387 tree automata pairs of different sizes generated from the intermediate steps of abstract regular tree model checking of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node [4].

The results of the experiments are presented in the following tables. Table 1 compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair. The results are grouped into two sets according to the size of the automata measured in the number of states. The table also gives the average speedup of the fastest upward approach compared to the fastest downward approach in case the upward computation was faster than the downward one (and vice versa). Table 2 provides a comparison of the methods that use simulation (either upward for `Up+s` or downward for `Down+s`) without counting the time for computing simulation (in such cases they were always faster than the methods not using simulations). This comparison is motivated by the observation that inclusion checking may be used as a part of a bigger computation that anyway computes the simulation relations (which happens, e.g., in abstract regular model checking where the simulations are used for reducing the size of the encountered automata). Finally, Table 3 summarizes how often the particular methods were successful in our testing runs (i.e., how often they did not timeout.).

Table 2. Percentages of cases in which the methods were the fastest when not counting the time for computing the simulation

Size	Pairs	Timeout	`Up+s`	`Down+s`	Avg up speedup	Avg down speedup
50–250	323	20 s	81.82 %	18.18 %	1.33	3.60
400–600	64	60 s	20.31 %	79.69 %	9.92	2116.29

Table 3. Percentages of successful runs that did not timeout

Size	Pairs	Timeout	`Up`	`Up+s`	`Down`	`Down+s`
50–250	323	20 s	100.00 %	100.00 %	74.92 %	99.07 %
400–600	64	60 s	51.56 %	51.56 %	39.06 %	90.62 %

The results show that the overhead of computing upward simulation is too high in all the cases that we have considered, causing upward inclusion checking using simulation to be the slowest when the time for computing the simulation used by the algorithm is included[6]. Next, it can be seen that for each of the remaining approaches there are cases in which they win in a significant way. However, the downward approaches are clearly dominating in significantly more of our test cases (with the only exception being the case of small automata when the time of computing simulations is not included). Moreover, the dominance of the downward checking increases with the size of the automata that we considered in our test cases.

4 Semi-Symbolic Representation of Tree Automata

We next consider a natural, semi-symbolic, MTBDD-based encoding of non-deterministic TA, suitable for handling automata with huge alphabets. We propose algorithms for computing downward simulations and for efficient downward inclusion checking on the considered representation. Due to space restrictions, we defer algorithms for further operations on the considered semi-symbolic representation of TA, including upward inclusion checking, to [12].

4.1 Binary Decision Diagrams

Let $\mathbb{B} = \{0,1\}$ be the set of Boolean values. A *Boolean function* of *arity* k is a function of the form $f : \mathbb{B}^k \to \mathbb{B}$. We extend the notion of Boolean functions to an arbitrary nonempty set S where a k-ary Boolean function extended to the domain set S is a function of the form $f : \mathbb{B}^k \to S$.

A *reduced ordered binary decision diagram* (ROBDD) [8] r over n Boolean variables $x_1, \ldots, x_n$ is a connected directed acyclic graph with a single *source node* (denoted as *r.root*) and at least one of the two *sink nodes* **0** and **1**. We call *internal* the nodes which are not sink nodes. A function *var* assigns each internal node a Boolean variable from the set $X = \{x_1, \ldots, x_n\}$, which is assumed to be ordered by the ordering $x_1 < x_2 < \cdots < x_n$. For every internal node v there exist 2 outgoing edges labelled *low* and *high*. We denote by *v.low* a node w and by *v.high* a node z such that there exists a directed edge from v to w labelled by *low* and a directed edge from v to z labelled by *high*, respectively. For each internal node v, it must hold that $var(v) < var(v.low)$ and $var(v) < var(v.high)$ and also $v.low \neq v.high$. A node v represents an n-ary Boolean function $[\![v]\!] : \mathbb{B}^n \to \mathbb{B}$ that assigns to each assignment to the Boolean variables in X a corresponding Boolean value defined in the following way (using $\bar{x}$ as an abbreviation for $x_1 \ldots x_n$): $[\![\mathbf{0}]\!] = \lambda\, \bar{x} \,.\, 0$, $[\![\mathbf{1}]\!] = \lambda\, \bar{x} \,.\, 1$, and $[\![v]\!] = \lambda\, \bar{x} \,.\, (\neg x_i \wedge [\![v.low]\!]) \vee (x_i \wedge [\![v.high]\!])$ for $var(v) = x_i$. For every two nodes v and w, it holds that $v \neq w \implies [\![v]\!] \neq [\![w]\!]$. We say that an ROBDD r represents the Boolean function $[\![r]\!] = [\![r.root]\!]$. Dually, for a Boolean function f, we use $\langle f \rangle$ to denote the ROBDD representing f, i.e., $f = [\![\langle f \rangle]\!]$.

We generalise the standard *Apply* operation for manipulation of Boolean functions represented by ROBDDs in the following way: let op_1, op_2, and op_3 be in turn arbitrary unary, binary, and ternary Boolean functions. Then the functions *Apply₁*, *Apply₂*, and *Apply₃* produce a new ROBDD which is defined as follows for ROBDDs f, g, and h: $\mathit{Apply_1}(f, op_1) = \langle \lambda\, \bar{x} \,.\, op_1([\![f(\bar{x})]\!]) \rangle$, $\mathit{Apply_2}(f, g, op_2) = \langle \lambda\, \bar{x} \,.\, op_2([\![f(\bar{x})]\!], [\![g(\bar{x})]\!]) \rangle$, and $\mathit{Apply_3}(f, g, h, op_3) = \langle \lambda\, \bar{x} \,.\, op_3([\![f(\bar{x})]\!], [\![g(\bar{x})]\!], [\![h(\bar{x})]\!]) \rangle$. In practice, one can also use *Apply* operations with side-effects.

[6] Note that `Up+s` was winning over `Up` in the experiments of [2] even with the time for computing simulation included, which seems to be caused by a much less efficient implementation of the antichains in the original algorithm.

The notion of ROBDDs is further generalized to *multi-terminal binary decision diagrams* (MTBDDs) [9]. MTBDDs are essentially the same data structures as ROBDDs, the only difference being the fact that the set of sink nodes is not restricted to two nodes. Instead, it can contain an arbitrary number of nodes labelled uniquely by elements of an arbitrary domain set S. All standard notions for ROBDDs can naturally be extended to MTBDDs. An MTBDD m then represents a Boolean function extended to S, i.e., $[\![m]\!] : \mathbb{B}^n \to S$. Further, the concept of *shared MTBDDs* is used. A shared MTBDD s is an MTBDD with multiple source nodes (or *roots*) that represents a mapping of every element of the set of roots R to a function induced by the MTBDD corresponding to the the given root, i.e., $[\![s]\!] : R \to (\mathbb{B}^n \to S)$.

4.2 Encoding The Transition Function of a TA Using Shared MTBDDs

We fix a tree automaton $\mathcal{A} = (Q, \Sigma, \Delta, F)$ for the rest of the section. We consider both a top-down and a bottom-up representation of its transition function. This is because some operations on $\mathcal{A}$ are easier to do on the former representation while others on the latter. We assume w.l.o.g. that the input alphabet Σ of $\mathcal{A}$ is represented in binary using n bits. We assign each bit in the binary encoding of Σ a Boolean variable from the set $\{x_1, \ldots, x_n\}$. We can then use shared MTBDDs with a set of roots R and a domain set S for encoding the various functions of the form $R \to (\Sigma \to S)$ that we will need.

Our *bottom-up* representation of the transition function Δ of the TA $\mathcal{A}$ uses a shared MTBDD Δ^{bu} over Σ where the set of root nodes is $Q^{\#}$, and the domain of labels of sink nodes is 2^Q. The MTBDD Δ^{bu} represents a function $[\![\Delta^{bu}]\!] : Q^{\#} \to (\Sigma \to 2^Q)$ defined as $[\![\Delta^{bu}]\!] = \lambda\ (q_1, \ldots, q_p)\ a\ .\ \{q \mid (q_1, \ldots, q_p) \xrightarrow{a} q\}$. It clearly holds that $[\![\Delta^{bu}((q_1, \ldots, q_p), a)]\!] = up_a((q_1, \ldots, q_p))$.

Our *top-down* representation of the transition function Δ of the TA $\mathcal{A}$ uses a shared MTBDD Δ^{td} over Σ where the set of root nodes is Q, and the domain of labels of sink nodes is $2^{Q^{\#}}$. The MTBDD Δ^{td} represents a function $[\![\Delta^{td}]\!] : Q \to (\Sigma \to 2^{Q^{\#}})$ defined as $[\![\Delta^{td}]\!] = \lambda\ q\ a\ .\ \{(q_1, \ldots, q_p) \mid q \xrightarrow{a} (q_1, \ldots, q_p)\}$. Clearly, $[\![\Delta^{td}(q, a)]\!] = down_a(q)$.

Sometimes it is necessary to convert between the bottom-up and top-down representation of a TA. For instance, when computing downward simulations (as explained below), one needs to switch between the bottom-up and top-down representation. Fortunately, the two representations are easy to convert (cf. [12]).

4.3 Downward Simulation on Semi-Symbolically Encoded TA

We next give an algorithm for computing the maximum downward simulation relation on the states of the TA $\mathcal{A}$ whose transition function is encoded using our semi-symbolic representation. The algorithm is inspired by the algorithm from [14] proposed for computing simulations on finite (word) automata. For use in the algorithm, we extend the notion of downward simulation to tuples of states by defining $(q_1, \ldots, q_n) \preceq_D (r_1, \ldots, r_n)$ to hold iff $\forall 1 \leq i \leq n : q_i \preceq_D r_i$.

Our algorithm for computing downward simulations, shown as Algorithm 3, starts with a gross over-approximation of the maximum downward simulation, which is then pruned until the maximum downward simulation is obtained. The algorithm uses the following main data structures:

- For each $q \in Q$, $sim(q) \subseteq Q$ is the set of states that are considered to simulate q at the current step of the computation. Its value is gradually pruned during the computation. At the end, it encodes the maximum downward simulation being computed.

Algorithm 3: Computing downward simulation on semi-symbolic TA

```
Input: Tree automaton $\mathcal{A} = (Q, \Sigma, \Delta^{bu}, F)$
Output: Maximum downward simulation $\preceq_D \subseteq Q^2$
/* initialization */
$\Delta^{td} :=$ invertMTBDD($\Delta^{bu}$);
$remove := \emptyset$;
$initCnt := \langle \lambda\ a\ .\ \emptyset \rangle$ ;                /* $[\![initCnt]\!] : \Sigma \to (Q \to \mathbb{N})$ */
foreach $q \in Q$ do
    $sim(q) := \emptyset$;
    $initCnt := Apply_2(\Delta^{td}(q), initCnt, (\lambda\ X\ Y\ .\ Y \cup \{(q, |X|)\})$;
    foreach $r \in Q$ do
        $isSim := true$;
        $Apply_2(\Delta^{td}(q), \Delta^{td}(r), (\lambda\ X\ Y$ . if $(X \neq \emptyset \wedge Y = \emptyset)$ then $isSim := false$)) ;
        if $isSim$ then
            $sim(q) := sim(q) \cup \{r\}$;
        else
            foreach $(q_1, \dots, q_n) \in Q^\#, (r_1, \dots, r_n) \in Q^\# : \exists 1 \leq i \leq n : q_i = q \wedge r_i = r$ do
                $remove := remove \cup \{((q_1, \dots, q_n), (r_1, \dots, r_n))\}$;
$cnt := \langle \lambda\ (q_1, \dots, q_n)\ a\ .\ \emptyset \rangle$ ;      /* $[\![cnt]\!] : Q^\# \to (\Sigma \to (Q \to \mathbb{N}))$ */
foreach $(q_1, \dots, q_n) \in Q^\#$ do $cnt((q_1, \dots, q_n)) := initCnt$;
/* computation */
while $\exists((q_1, \dots, q_n), (r_1, \dots, r_n)) \in remove$ do
    $remove := remove \setminus \{((q_1, \dots, q_n), (r_1, \dots, r_n))\}$;
    $cnt((q_1, \dots, q_n)) :=$
    $Apply_3(\Delta^{bu}((r_1, \dots, r_n)), \Delta^{bu}((q_1, \dots, q_n)), cnt((q_1, \dots, q_n)),$ (refine $sim\ remove$));
return $\{(q, r) \mid q \in Q, r \in sim(q)\}$;
```

Function refine(*&sim, &remove, up$_a$R, up$_a$Q, cnt$_a$Q*)

```
$newCnt_aQ := cnt_aQ$;
foreach $s \in up_aR$ do
    $newCnt_aQ(s) := newCnt_aQ(s) - 1$;
    if $newCnt_aQ(s) = 0$ then
        foreach $p \in up_aQ : s \in sim(p)$ do
            foreach $(p_1, \dots, p_n) \in Q^\#, (s_1, \dots, s_n) \in Q^\# : \exists 1 \leq i \leq n : p_i = p \wedge s_i = s$ do
                if $\forall 1 \leq j \leq n : s_j \in sim(p_j)$ then
                    $remove := remove \cup \{((p_1, \dots, p_n), (s_1, \dots, s_n))\}$;
            $sim(p) := sim(p) \setminus \{s\}$;
return $newCnt_aQ$;
```

– The set *remove* $\subseteq Q^\# \times Q^\#$ contains pairs $((q_1, \dots, q_n), (r_1, \dots, r_n))$ of tuples of states, for which it is known that $(q_1, \dots, q_n) \not\preceq_D (r_1, \dots, r_n)$, for processing.
– Finally, *cnt* is a shared MTBDD encoding a function $[\![cnt]\!] : Q^\# \to (\Sigma \to (Q \to \mathbb{N}))$ that for each $(q_1, \dots, q_n) \in Q^\#$, $a \in \Sigma$, and $q \in Q$, gives a value $h \in \mathbb{N}$ such that $(q_1, \dots, q_n)$ can make a bottom-up transition over a to h distinct states $r \in sim(q)$.

The algorithm works in two phases. We assume that we start with a TA whose transition function is represented bottom-up. In the *initialization* phase, the dual top-down representation of the transition function is first computed (note that we can also start with a top-down representation and compute the bottom-up representation as both are needed in the algorithm). The three main data structures are then initialized as follows:

– For each $q \in Q$, the set $sim(q)$ is initialized as the set of states that can make top-down transitions over the same symbols as q, which is determined using the *Apply* operation on line 9. This is, when starting the main computation loop on line 17, the value of *sim* for each state $q \in Q$ is $sim(q) = \{r \mid \forall a \in \Sigma : q \xrightarrow{a} (q_1,\dots,q_n) \implies r \xrightarrow{a} (r_1,\dots,r_n)\}$.

- The *remove* set is initialized to contain each pair of tuples of states $((q_1,\dots,q_n), (r_1,\dots,r_n))$ for which it holds that the relation $(q_1,\dots,q_n) \preceq_D (r_1,\dots,r_n)$ is broken even for the initial approximation of $\preceq_D$, i.e., for some position $1 \le i \le n$ there is a pair $q_i, r_i \in Q$ such that $r_i \notin sim(q_i)$.
- To initialize the shared MTBDD *cnt*, the algorithm constructs an auxiliary MTBDD *initCnt* representing a function $[\![initCnt]\!] : \Sigma \to (Q \to \mathbb{N})$. Via the *Apply* operation on line 6, this MTBDD gradually collects, for each symbol $a \in \Sigma$, the set of pairs (q,h) such that q can make a top-down transition to h distinct tuples over the symbol a. This MTBDD is then copied to the shared MTBDD *cnt* for each tuple of states $(q_1,\dots,q_n) \in Q^{\#}$. This is justified by the fact that we start by assuming that the simulation relation is equal to $Q \times Q$, which for a symbol $a \in \Sigma$ and a pair $(q,h) \in cnt((q_1,\dots,q_n))$ means that $(q_1,\dots,q_n)$ can make a bottom-up transition over a to h distinct states $r \in sim(q)$.

The main *computation* phase gradually restricts the initial over-approximation of the maximum downward simulation being computed. As we have said, the *remove* set contains pairs $((q_1,\dots,q_n),(r_1,\dots,r_n))$ for which it holds that $(q_1,\dots,q_n)$ cannot be simulated by $(r_1,\dots,r_n)$, i.e., $(q_1,\dots,q_n) \not\preceq_D (r_1,\dots,r_n)$. When such a pair is processed, the algorithm decrements the counter $[\![cnt((q_1,\dots,q_n),a,s)]\!]$ for each state s for which there exists a bottom-up transition over a symbol $a \in \Sigma$ such that $(r_1,\dots,r_n) \xrightarrow{a} s$. The meaning is that s can make one less top-down transition over a to some $(t_1,\dots,t_n)$ such that $(q_1,\dots,q_n) \preceq_D (t_1,\dots,t_n)$. If $[\![cnt((q_1,\dots,q_n),a,s)]\!]$ drops to zero, it means that s cannot make a top-down transition over a to any $(t_1,\dots,t_n)$ such that $(q_1,\dots,q_n) \preceq_D (t_1,\dots,t_n)$. This means, for all $p \in Q$ such that p can make a top-down transition over a to $(q_1,\dots,q_n)$, that s no longer simulates p, i.e., $p \not\preceq_D s$. When the simulation relation between p and s, $p \preceq_D s$, is broken, then the simulation relation between all m-tuples $(p_1,\dots,p_m)$ and $(s_1,\dots,s_m)$ such that $\exists 1 \le j \le m : p_j = p \wedge s_j = s$ must also be broken, therefore the pair $((p_1,\dots,p_m),(s_1,\dots,s_m))$ is put to the *remove* set (unless the simulation relation between some other states in the tuples has already been broken before).

Correctness of the algorithm is summarised in the below theorem, which can be proven analogically as correctness of the algorithm proposed in [14], taking into account the meaning of the above described MTBDD-based structures and the operations performed on them.

Theorem 4. *When applied on a TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ whose transition function is encoded semi-symbolically in the bottom-up way as Δ^{bu}, Algorithm 3 terminates and returns the maximum downward simulation on Q.*

4.4 Downward Inclusion Checking on Semi-Symbolically Encoded TA

We now proceed to an algorithm of efficient downward inclusion checking on semi-symbolically represented TA. The algorithm we propose for this purpose is derived from Algorithm 2 by plugging the `expand3` function instead of the `expand2` function. It is based on the same basic principle as `expand2`, but it has to cope with the symbolically encoded transition relation. In particular, in order to inspect whether for a pair (p_S, P_B) and all symbols $a \in \Sigma$ the inclusion between each tuple from $down_a(p_S)$ and the set of

Function `expand3` (p_S, P_B, *workset*)

```
/* $p_S \in Q_S$, $P_B \subseteq Q_B$, and $workset \subseteq Q_S \times 2^{Q_B}$ */
if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ then return true;
if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ then return false ;
if $\exists p \in P_B : p_S \preceq p$ then return true;
$workset := workset \cup \{(p_S, P_B)\}$;
$tmp := \langle \lambda\, a\, .\, \emptyset \rangle$;
foreach $p_B \in P_B$ do
    $tmp := Apply_2(tmp, \Delta^{td}_B(p_B), (\lambda\, X\, Y\, .\, X \cup Y))$;
$doesInclusionHold := true$;
$Apply_2(\Delta^{td}_S(p_S), tmp, ($procDown $doesInclusionHold\ workset))$;
return $doesInclusionHold$;
```

Function `procDown` (*&doesInclusionHold, &workset, $down_a p_S$, $down_a P_B$*)

```
if $() \in down_a p_S \wedge () \notin down_a P_B$ then
    $doesInclusionHold := false$;
else
    $W := down_a P_B$;
    foreach $(r_1, \ldots, r_n) \in down_a p_S$ do          /* $p_S \xrightarrow{a} (r_1, \ldots, r_n)$ */
        foreach $f \in \{W \to \{1, \ldots, n\}\}$ do
            $found := false$;
            foreach $1 \leq i \leq n$ do
                $S := \{q_i \mid (q_1, \ldots, q_n) \in W, f((q_1, \ldots, q_n)) = i\}$;
                if expand3$(r_i, S, workset)$ then
                    $found := true$;
                    break;
                if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ then
                    $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
            if $\neg found$ then
                $doesInclusionHold := false$;
                return;
```

tuples $down_a(P_B)$ holds, the *doesInclusionHold* parameter initialized to *true* is passed to the *Apply* operation on line 9 of the `expand3` function. If the algorithm finds out that the inclusion does not hold in some execution of the `procDown` function in the context of a single *Apply*, *doesInclusionHold* is assigned the *false* value, which is later returned by `expand3`. Otherwise `expand3` returns its original *true* value.

4.5 Experimental Results

We have implemented a prototype of a library for working with TA encoded semi-symbolically as described above. We have used the CUDD library [17] as an implementation of shared MTBDDs. The prototype contains the algorithms presented in this section and some more presented in [12]. The results on downward inclusion checking that we have obtained with the explicitly represented TA encouraged us to also compare performance of the upward inclusion checking and downward inclusion checking on automata with large alphabets using our prototype.

We have compared the upward inclusion checking algorithm from [4] adapted for semi-symbolically represented tree automata, which is given in [12] (and marked as `UpSym` in the following), with the downward inclusion checking algorithm presented above. In the latter case, we let the algorithm use either the identity relation, which corresponds to downward inclusion checking without using any simulation (this case is marked as `DownSym` below), or the maximum downward simulation (which is marked as `DownSym+s` in the results). We have not considered upward inclusion checking with upward simulation due to its negative results in our experiments with explicitly encoded automata[7]. For the comparison, we used 97 pairs of tree automata with a large alphabet which we encoded into 12 bits. The size of the automata was between 50 and 150 states and the timeout was set to 300 s. The automata were obtained by taking the automata considered in Section 3.3 and labelling their transitions by randomly generated sets of symbols from the considered large alphabet.

The results that we have obtained are presented in the following tables. Table 4 compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair. This table also presents the average speedup of the upward approach compared to the fastest downward approach in case the upward computation was faster than the downward one (and vice versa). Table 5 summarizes how often each of the methods was successful in the testing runs.

Table 4. Percentages of cases in which the respective methods were the fastest

`UpSym`	`DownSym`	`DownSym+s`	Avg up speedup	Avg down speedup
6.67 %	90.67 %	2.67 %	24.39	4389.76

Table 5. Successful runs that did not timeout (in %)

`UpSym`	`DownSym`	`DownSym+s`
77.32 %	77.32 %	26.08 %

When we compare the above experimental results with the results obtained on the explicitly represented automata presented in Section 3.3, we may note that (1) downward inclusion checking is again significantly dominating, but (2) the advantage of exploiting downward simulation has decreased. According to the information we gathered from code profiling of our implementation, this is due to the overhead of the CUDD library which is used as the underlying layer for storage of shared MTBDDs of several data structures (which indicates a need of a different MTBDD library to be used or perhaps of a specialised MTBDD library to be developed).

We also evaluated performance of the implementation of the described algorithms using a semi-symbolic encoding of TA with performance of the algorithms using an explicit encoding of TA considered in Section 3 on the automata with the large alphabet. The symbolic version was in average 8676 times faster than the explicit one as expected when using a large alphabet.

5 Conclusion

We have proposed a new algorithm for checking language inclusion over non-deterministic TA (based on the one from [13]) that traverses automata in the downward manner and uses both antichains and simulations to optimize its computation. This algorithm is, according to our experimental results, mostly superior to the known upward algorithms. We have further presented a semi symbolic MTBDD-based representation of non-deterministic TA generalising the one used by MONA, together with important tree automata algorithms working over this representation, most notably an algorithm

[7] We, however, note that possibilities of implementing upward inclusion checking combined with upward simulations over semi-symbolically encoded TA and a further evaluation of this algorithm are still interesting subjects for the future.

for computing downward simulations over TA inspired by [14] and the downward language inclusion algorithm improved by simulations and antichains proposed in this paper. We have experimentally justified usefulness of the symbolic encoding for nondeterministic TA with large alphabets.

Our experimental results suggest that the MTBDD package CUDD is not very efficient for our purposes and that better results could probably be achieved using a specialised MTBDD package whose design is an interesting subject for further work. Apart from that, it would be interesting to encode antichains used within the language inclusion checking algorithms symbolically as, e.g., in [18]. An interesting problem here is how to efficiently encode antichains based not on the subset inclusion but on a simulation relation. Finally, as a general target, we plan to continue in our work towards obtaining a really efficient TA library which could ultimately replace the one of MONA.

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*, LNCS 5148, Springer, 2008.
2. P. A. Abdulla, L. Holík, Y.-F. Chen, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In *Proc. of TACAS'10*, LNCS 6015, Springer, 2010.
3. P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, LNCS 2404, Springer, 2002.
4. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In *Proc. of CIAA'08*, LNCS 5148, Springer, 2008.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149, Elsevier, 2006.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, LNCS 4134, Springer, 2006.
7. T. Bourdier. Tree Automata-based Semantics of Firewalls. In *Proc. of SAR-SSI'11*, IEEE, 2011.
8. R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 1986.
9. E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. *FMSD*, 10, Springer, 1997.
10. L. Doyen and J. F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. of TACAS'10*, LNCS 6015, Springer, 2010.
11. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011
12. L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. Tech. rep. FIT-TR-2011-04, FIT BUT, Czech Rep., 2011.
13. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.*, 27, 2005.
14. L. Ilie, G. Navarro, and S. Yu. On NFA Reductions. In *Proc. of Theory is Forever*, LNCS 3113, Springer, 2004.
15. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA Implementation Secrets. *International Journal of Foundations of Computer Science*, 13(4), 2002.
16. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. *SIGPLAN Not.*, 46, 2011.
17. F. Somenzi. CUDD: CU Decision Diagram Package Release 2.4.2, May 2011.
18. A. Tozawa and M. Hagiya. XML Schema Containment Checking Based on Semi-implicit Techniques. In *Proc. of CIAA'03*, LNCS 2759, Springer, 2003.
19. M. De Wulf, L. Doyen, T. A. Henzinger, J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV'06*, LNCS 4144, Springer, 2006.