International Journal of Foundations of Computer Science

Bisimulation Minimization of Tree Automata

PAROSH AZIZ ABDULLA
Dept. of Information Technology, Uppsala University, P.O. Box 337
S–751 05 Uppsala, Sweden.
parosh@it.uu.se

and

JOHANNA HÖGBERG
Department of Computing Science, Umeå University
S–901 87 Umeå, Sweden
johanna@cs.umu.se

and

LISA KAATI
Dept. of Information Technology, Uppsala University, P.O. Box 337
S–751 05 Uppsala, Sweden.
lisa.kaati@it.uu.se

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We extend an algorithm by Paige and Tarjan that solves the coarsest stable refinement problem to the domain of trees. The algorithm is used to minimize non-deterministic tree automata (NTA) with respect to bisimulation. We show that our algorithm has an overall complexity of $O(\hat{r}\, m \log n)$, where $\hat{r}$ is the maximum rank of the input alphabet, m is the total size of the transition table, and n is the number of states.

Keywords: bisimulation, tree automata, minimization

1. Introduction

We present an algorithm that minimizes non-deterministic tree automata with respect to bisimulation equivalence in time $O(\hat{r}\, m \log n)$, where $\hat{r}$ is the maximum rank of the input alphabet, m is the total size of the transition table, and n is the number of states. In the construction of this algorithm, we extend an algorithm by Paige and Tarjan proposed in [18] to the domain of trees. Since the time complexity reduces to $O(m \log n)$ if $\hat{r}$ is constant, this retains the complexity of [18] in all cases where the maximum rank of the input alphabet is bounded. This holds in particular for monadic trees, i.e. the string case.

The minimization of finite *string* automata (FA) is a well-studied problem, where the objective is to find the unique minimal FA that recognizes the same language as a given FA. In the deterministic case, efficient algorithms are available, e.g. the algorithm proposed by Hopcroft in [11], where he uses a “process the smaller half” strategy to obtain a bound of $O(n \log n)$. However, it has been proven that minimization of non-deterministic finite automata (FA) is PSPACE complete [15], and what is worse, that the minimization problem for an NFA with n states cannot be efficiently approximated within the factor $o(n)$, unless P = PSPACE [10]. To avoid exponential time, the problem must either be restricted (e.g. by considering a special class of devices or requiring additional information), or no approximation guarantees can be given. Of course, this holds also for non-deterministic tree automata (NTA) because they generalize NFAs (as a string may be seen as a monadic tree). Hence, we cannot hope to find an efficient algorithm that performs well on all input NTAs.

The algorithm presented in this paper takes advantage of the fact that bisimulation equivalence is computationally easier to decide than language equivalence, and that bisimulation equivalence implies language equivalence (although the converse does not hold in the general case) [13]. When minimizing an NTA, we group states that are observationally equivalent and use the blocks of the resulting partition as states in the output NTA. As mentioned above, the time complexity becomes $O(\hat{r}\ m \log n)$, as computed to $O(m \log n)$ by [18]. Interestingly, the maximum rank $\hat{r}$ (which is the constant 1 in [18]) does not become an exponent. Instead, it influences the complexity rather modestly.

Bisimulation minimization of tree automata is of particular interest in *tree regular model checking* (an extension of regular model checking [3]). In this field, the verification of infinite state systems with tree-like architecture is considered, and many of the associated algorithms would benefit from an efficient method to reduce the size of tree automata [2].

Related Work To the best of our knowledge, there is no documented algorithm that uses bisimulation to minimize NTA, but we do know of a number of minimization algorithms that operate on various kinds of tree automata. For instance, algorithms for *guided tree automata* (GTA) are considered in [5]. A GTA is a bottom-up tree automaton equipped with separate state spaces that are assigned by a top-down automaton. According to the authors, minimization of GTA is possible in time $O(nm)$, (here, n is the total number of states and m the total representation size) but it still remains an open question whether or not tree automata can be minimized in time $O(m \log m)$. In [9], Cristau et al. claim that a deterministic bottom-up tree automaton for unranked trees can be minimized in quadratic time using the algorithm proposed in [11]. In [6], the minimal tree automaton is computed using an algorithm that construct congruences of the input automaton until a fixed point is reached, but no results regarding the computational complexity are presented.

In [17], Nivat and Podelski propose a generalization of deterministic top-down tree automata to *r-l-deterministic* top-down tree automata (l-r DTA). The recognizing power of this device lies strictly between that of "traditional" deterministic and non-deterministic top-down tree automata. Nivat and Podelski shows every l-r DTA A can be minimized to obtain the unique minimal l-r DTA recognizing $L(A)$, although they do not formulate an explicit minimization algorithm. There have also been attempts to impose restrictions on NTAs to obtain devices with nicer mathematical properties. In [8], a new class of non-deterministic tree automata called *residual finite tree automata* (RFTA) is defined, where the bottom-up case gives a new characterization of regular tree languages. The authors state that for a given RFTA, there always exists a canonical NTA (the problem of finding it is still in PSPACE).

Outline Section 2 covers the preliminaries, while Section 3 generalizes a partitioning algorithm from [18] to trees. Section 4 describes the necessary calculation steps. In Section 5, the extended algorithm is applied to the minimization of NTA, and in Section 6, we show experimental results obtained from a prototype. We conclude with some directions for future work.

2. Preliminaries

Tree Automata A *ranked alphabet* is a finite set of symbols $\Sigma = \bigcup_{k \in \mathbb{N}} \Sigma_{(k)}$ which is partitioned into pairwise disjoint subsets $\Sigma_{(k)}$. The symbols in $\Sigma_{(k)}$ are said to have *rank* k. The set T_Σ of all *trees over* Σ is the smallest superset of $\Sigma_{(0)}$ that contains every $f\,[t_1, \ldots, t_k]$, where $f \in \Sigma_{(k)}$, $k \geq 1$, and $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$. A subset of T_Σ is called a *tree language*.

A *bottom-up tree automaton* is a quadruple $A = (\Sigma, Q, \delta, F)$ where Σ is a ranked input alphabet, Q is a finite set of *states*, δ is a finite set of *transition rules* of the form $f(q_1, \ldots, q_n) \to q_{n+1}$ where $f \in \Sigma_{(n)}$, and $q_1, \ldots, q_{n+1} \in Q$, for some $n \in \mathbb{N}$. In the special case when δ is a function, we say that the automaton is *deterministic*. Finally, $F \subseteq Q$ is a set of *accepting* states.

The relation δ easily extends to trees, yielding a function $\delta \colon \mathrm{T}_\Sigma \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ denotes the powerset of Q: for $t = f[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$, we let

$$\delta(t) = \{q \mid f\,(q_1, \ldots, q_k) \to q \in \delta \text{ and } q_i \in \delta(t_i) \text{ for all } i : 1 \leq i \leq k\} \enspace .$$

The tree language *recognized* by an automaton A is $L(A) = \{t \in \mathrm{T}_\Sigma \mid \delta(t) \cap F \neq \emptyset\}$.

Note that a deterministic bottom-up tree automaton A with a set of transition rules δ maps every tree in T_Σ to a singleton set. In other words, in the deterministic case δ is a function from T_Σ to the states of A.

Let $r = f\,(q_1, \ldots, q_n) \to q_{n+1}$ be a transition rule, then $|r|$ denotes its length (i.e., $|r| = n{+}1$), $r(i)$ denotes the state q_i, and $q \in r$ indicates that $r(i) = q$ for some $i : 1 \leq i \leq |r|$. For $B \subseteq Q$, take δ_B as the set $\{r \in \delta \mid \exists q \in B \text{ such that } q \in r\}$.

Equivalences We consider equivalence relations on Q. Let $\simeq'$ and $\simeq$ be two equivalence relations such that $\simeq' \subseteq \simeq$ (i.e., $\simeq'$ is a *refinement* of $\simeq$). We write $(Q/{\simeq})$ to denote the set of equivalence classes (henceforth, *blocks*) of $\simeq$, and $[q]_\simeq$ to denote unique the block of $\simeq$ that contains q. For a block $B \in (Q/{\simeq})$, we write

$[\![B]\!]_{\simeq'}$ to denote the set $\{B' \in (Q/\simeq') \mid B' \subseteq B\}$. For a block $B' \in (Q/\simeq')$, we let $[B']_{\simeq}$ represent the (unique) block $B \in (Q/\simeq)$ such that $B' \in [\![B]\!]_{\simeq'}$.

Symbolic rules Let $A = (Q, \Sigma, \delta, F)$ be an NTA, and let $\simeq'$ and $\simeq$ be equivalence relations on Q such that $\simeq'$ is a refinement of $\simeq$. We use the notation $(\delta/\simeq)$ to represent the set

$$\{f([q_1]_{\simeq}, \ldots, [q_k]_{\simeq}) \rightarrow [q]_{\simeq} \mid f(q_1, \ldots, q_k) \rightarrow q \in \delta\}$$

of *symbolic rules with respect to* $\simeq$. Conversely, if $\rho = (D_1, \ldots, D_k) \rightarrow D_{k+1}$ is a symbolic rule, then the set of *instances* of ρ, denoted $[\![\rho]\!]$, is the set $\{(q_1, \ldots, q_k) \rightarrow q_{k+1} \mid q_i \in D_i,\ i : 1 \leq i \leq k+1\} \cap \delta$. We write $\rho(i)$ to refer to block D_i of ρ, and $B \in \rho$ to indicate that $\rho(i) = B$, for some $i : 1 \leq i \leq |\rho|$. For a block $B \in (Q/\simeq)$, we use δ_B to denote the set $\{\rho \in (\delta/\simeq) \mid B \in \rho\}$. The length of ρ is written $|\rho|$. For a transition rule $r \in \delta$, we use $[r]_{\simeq}$ to represent the unique symbolic rule $\rho \in (\delta/\simeq)$ such that $r \in [\![\rho]\!]$. Given a rule $\rho = (D_1, ..., D_n) \rightarrow D_{n+1}$ in $(\delta/\simeq)$, we let $[\![\rho]\!]_{\simeq'}$ represent the set

$$\{(D'_1, ..., D'_n) \rightarrow D'_{n+1} \in (\delta/\simeq') \mid D'_i \in [\![D_i]\!]_{\simeq'},\ \text{for all } i : 1 \leq i \leq n+1\} \ .$$

We write $[\![\rho]\!]^B_{\simeq'}$ to denote the subset

$$\left\{(D'_1, ..., D'_n) \rightarrow D'_{n+1} \mid \exists i \in \{1, \ldots, n+1\} \text{ s.t. } D'_i = B\right\}$$

of $[\![\rho]\!]_{\simeq'}$. Conversely, for the symbolic rule $\rho' \in (\delta/\simeq')$, we define $[\rho']_{\simeq}$ to be the (unique) symbolic rule $\rho \in (\delta/\simeq)$ such that $\rho' \in [\![\rho]\!]_{\simeq'}$.

Occurrences and Counts Let $\simeq$ be an equivalence relation and q a state, then the set of *occurrences* of q in $\simeq$, denoted $Occ(\simeq)(q)$, is the set of pairs (ρ, j) such that $j : 1 \leq j \leq |\rho|$ and there is an $r \in [\![\rho]\!]_{\simeq}$ with $q = \rho(j)$. Intuitively, $Occ(\simeq)(q)$ identifies the symbolic rules in which $[q]$ occurs in the rule, together with the position of such an occurrence. Given a block B in $\simeq$, we define $Occ(\simeq)(B)(q)$, to be

$$\{(\rho, i) \mid (\rho, i) \in Occ(\simeq)(q) \text{ and } B \in \rho\}$$

For a symbolic rule ρ, a state q, and $j : 1 \leq j \leq |\rho|$, we define $count(j)(\rho)(q)$ to be the size of the set $\{r \in [\![\rho]\!] \mid r(j) = q\}$. We extend the definition to a set ϱ of symbolic rules such that $count(j)(\varrho)(q) = \sum_{\rho \in \varrho} count(j)(\rho)(q)$.

Stability Let $\simeq$ and $\cong$, where $\simeq \subseteq \cong$, be equivalence relations on Q. The relation $\simeq$ is *stable with respect to* $\cong$ if whenever $q \simeq p$ then $Occ(\cong)(q) = Occ(\cong)(p)$, and *stable* if it is stable with respect to itself.

3. The Algorithm

In this section, we introduce an algorithm for solving the *coarsest stable refinement problem* for NTAs. An instance of the problem consists of an NTA A and an equivalence relation $\simeq_{init}$ on the states of A. The task is to find the *stable* (as defined in the previous section) refinement $\simeq$ of $\simeq_{init}$ that is *coarsest* in the sense that every other stable refinement of $\simeq_{init}$ is also a refinement of $\simeq$.

The algorithm iterates over a sequence of steps (described in detail in Section 4) given an initial equivalence $\simeq_{init}$. The initial equivalence $\simeq_{init}$ is assumed to be a proper refinement of $Q \times Q$ which is stable with respect to $Q \times Q$, and which satisfies the condition that $q \simeq_{init} q' \Leftrightarrow q, q' \in F$.

The iterations generates two sequences of equivalence relations on Q, denoted by $\simeq_0, \simeq_1, \ldots, \simeq_t$ and $\cong_0, \cong_1, \ldots, \cong_t$ respectively. We define $\simeq_0$ to be $\simeq_{init}$ and $\cong_0$ to be $Q \times Q$.

In the $(i+1)$-th iteration, the equivalences $\simeq_{i+1}$ and $\cong_{i+1}$ are derived from $\simeq_i$ and $\cong_i$ as follows. Let $B_i \in (Q/\simeq_i)$ and $S_i \in (Q/\cong_i)$ be such that $B_i \subset S_i$ and $|B_i| \leq \frac{|S_i|}{2}$ (as implied by Lemma 1, $\simeq_i$ is a proper refinement of $\cong_i$ so B_i and S_i exist). We have that $q \cong_{i+1} q'$ if and only if two conditions are met. First, $q \cong_i q'$, and second, $q \in B_i$ if and only if $q' \in B_i$. Furthermore, for all $q, q' \in Q$, it holds that $q \simeq_{i+1} q'$ if and only if the following conditions are satisfied:

(i) $q \simeq_i q'$

(ii) $Occ(\cong_{i+1})(B_i)(q) \; = \; Occ(\cong_{i+1})(B_i)(q')$

(iii) For every $\rho \in (\delta/\cong_i)$, we have that $count(j)(\rho)(q) = count(j)\left(\llbracket\rho\rrbracket^{B_i}_{\cong_{i+1}}\right)(q)$ iff $count(j)(\rho)(q') = count(j)\left(\llbracket\rho\rrbracket^{B_i}_{\cong_{i+1}}\right)(q')$.

Intuitively, the second and third conditions refine $\simeq_i$ with respect to B_i and $S_i - B_i$ respectively. The iteration continues until we reach the termination point t, at which we have $\simeq_t = \cong_t$.

Correctness and time complexity We now argue that the algorithm is correct and runs in time $O(\hat{r}\, m \log n)$, beginning with a simple lemma.

Lemma 1 *The relation $\simeq_i$ is a refinement of $\cong_i$, for all $i : 0 \leq i \leq t$.*

Proof. By induction on i. The base case is trivial since $\cong_0 = Q \times Q$. Suppose that $q \simeq_{i+1} q'$. By definition of $\simeq_{i+1}$ it follows that $q \simeq_i q'$. By the induction hypothesis it follows that $q \cong_i q'$. Since $p \simeq_i q$ and $B_i \in (Q/\simeq_i)$ it follows that $q \in B_i$ iff $q' \in B_i$. By definition of $\cong_{i+1}$ it follows that $q \cong_{i+1} q'$. □

This implies that $\simeq_i$ is a *proper* refinement of $\cong_i$ for all $i : 0 \leq i < t$, and that, up to the termination point, we will be able to pick $B_i \in (Q/\simeq_i)$ and $S_i \in (Q/\cong_i)$ such that $B_i \subset S_i$ and $|B_i| \leq \frac{|S_i|}{2}$.

Next, we consider partial correctness of the algorithm which will follow from Lemma 3 and Lemma 5. To do that, we use the following auxiliary lemma (the proof can be found in [1]).

Lemma 2 *Consider equivalence relations $\simeq' \subseteq \simeq$, a symbolic rule $\rho \in (\delta/\simeq)$, a state q, and $j : 1 \leq j \leq |\rho|$. Then, we have $(\rho, j) \in Occ(\simeq)(q)$ if and only if $(\rho', j) \in Occ(\simeq')(q)$ for some $\rho' \in \llbracket\rho\rrbracket_{\simeq'}$.*

Lemma 3 *The relation $\simeq_i$ is stable with respect to $\cong_i$, for all $i : 1 \leq i \leq t$.*

Proof. By induction on i. The base case (when $i = 0$) follows from the definitions of $\simeq_0$ and $\cong_0$. Suppose that $q \simeq_{i+1} q'$, and that $(\rho, j) \in Occ(\cong_{i+1})(q)$ for some $\rho \in (\delta/\cong_{i+1})$; we show that $(\rho, j) \in Occ(\cong_{i+1})(q')$. Depending on ρ, we have three cases:

(i) $S_i - B_i \notin \rho$ and $B_i \notin \rho$. This means that $\rho \in (\delta/\cong_i)$, and hence by Lemma 2 we have that $(\rho, j) \in \mathit{Occ}(\cong_i)(q)$. Since $q \simeq_{i+1} q'$, we know by definition that $q \simeq_i q'$. By the induction hypothesis it follows that $\simeq_i$ is stable with respect to $\cong_i$, and hence $(\rho, j) \in \mathit{Occ}(\cong_i)(q')$. Since $\rho \in (\delta/\cong_{i+1})$ it follows by Lemma 2 that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(q')$.

(ii) $S_i - B_i \in \rho$ and $B_i \notin \rho$. Let ρ be of the form $(D_1, \ldots, D_n) \to D_{n+1}$. Define $\rho^1 \in (\delta/\cong_i)$ to be the symbolic rule $\left(D_1^1, \ldots, D_n^1\right) \to D_{n+1}^1$ where, for each $k : 1 \leq k \leq n+1$, we have that $D_k^1 = S_i$ if $D_k = S_i - B_i$ and $D_k^1 = D_k$ otherwise. We observe that $\rho^1 = [\rho]_{\cong_i}$, and hence by Lemma 2 we have that $\left(\rho^1, j\right) \in \mathit{Occ}(\cong_i)(q)$. Since $q \simeq_{i+1} q'$, we know by definition that $q \simeq_i q'$. By the induction hypothesis it follows that $\simeq_i$ is stable with respect to $\cong_i$, and hence $\left(\rho^1, j\right) \in \mathit{Occ}(\cong_i)(q')$. From $B_i \notin \rho$ we know that $\mathit{count}(j)(\rho^1)(q) > \mathit{count}(j)\left(\llbracket\rho^1\rrbracket_{\cong_{i+1}}^{B_i}\right)(q)$. Since $q \simeq_{i+1} q'$ it follows by definition of $\cong_{i+1}$ that $\mathit{count}(j)(\rho^1)(q') > \mathit{count}(j)\left(\llbracket\rho^1\rrbracket_{\cong_{i+1}}^{B_i}\right)(q')$. Hence, $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(q')$.

(iii) $B_i \in \rho$. This means that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(B_i)(q)$. Since $q \simeq_{i+1} q'$ it follows that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(B_i)(q')$ and hence $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(q)$.

□

In the proof of Lemma 5, we use the following auxiliary lemma (the proof can be found in [1]).

Lemma 4 *Any refinement $\simeq$ of $\simeq_i$, is also a refinement of $\cong_{i+1}$.*

Lemma 5 *If $\simeq$ is a stable refinement of $\simeq_0$, then $\simeq$ is also a stable refinement of $\simeq_i$, for each $i : 1 \leq i \leq t$.*

Proof. By induction on i. The base case is trivial. For the induction step, suppose that $q \simeq q'$. We show that $q \simeq_{i+1} q'$ using the three conditions in the definition of $\simeq_{i+1}$. Condition (1) is satisfied by the induction hypothesis.

For Condition (2), suppose that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(B_i)(q)$. Since (ρ, j) is in $\mathit{Occ}(\cong_{i+1})(B_i)(q)$ we know that $B_i \in \rho$ and that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(q)$. From the induction hypothesis we know that $\simeq \subseteq \simeq_i$, and by Lemma 1 and Lemma 4 that $\simeq \subseteq \cong_{i+1}$. By Lemma 2 there is a $\rho' \in \llbracket\rho\rrbracket_{\simeq}$ such that $(\rho', j) \in \mathit{Occ}(\simeq)(q)$. Since $q \simeq q'$ and $\simeq$ is stable, we have that $(\rho', j) \in \mathit{Occ}(\simeq)(q')$. From $\simeq \subseteq \cong_{i+1}$ and $\rho' \in \llbracket\rho\rrbracket_{\simeq}$, it follows by Lemma 2 that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(q')$. Since $B_i \in \rho$, we conclude that $(\rho, j) \in \mathit{Occ}(\cong_{i+1})(B_i)(q')$.

Regarding Condition (3), assume that $\mathit{count}(j)(\rho)(q) \neq \mathit{count}(j)(\llbracket\rho\rrbracket_{\cong_{i+1}}^{B_i})(q)$ for some $\rho \in (\delta/\cong_i)$. We show that $\mathit{count}(j)(\rho)(q') \neq \mathit{count}(j)(\llbracket\rho\rrbracket_{\cong_{i+1}}^{B_i})(q')$. From the above assumption, we know that $S_i \in \rho$, $(\rho, j) \in \mathit{Occ}(\cong_i)(q)$, and that there is a $\rho_1 \in \llbracket\rho\rrbracket_{\cong_{i+1}}$ such that $B_i \notin \rho_1$ and $(\rho_1, j) \in \mathit{Occ}(\cong_{i+1})(q)$. Form the induction hypothesis we know that $\simeq \subseteq \simeq_i$, and hence by Lemma 4 it follows that $\simeq \subseteq \cong_{i+1}$. By Lemma 2 there is a $\rho_2 \in \llbracket\rho_1\rrbracket_{\simeq}$, such that $(\rho_2, j) \in \mathit{Occ}(\simeq)(q)$. Since $q \simeq q'$ and $\simeq$ is stable it follows that $(\rho_2, j) \in \mathit{Occ}(\simeq)(q')$. From $\simeq \subseteq \cong_{i+1}$ and $\rho_2 \in \llbracket\rho_1\rrbracket_{\simeq}$, it follows by Lemma 2 that $(\rho_1, j) \in \mathit{Occ}(\cong_{i+1})(q')$. Since $\rho_1 \in \llbracket\rho\rrbracket_{\cong_{i+1}}$ it follows by Lemma 2 that $(\rho, j) \in \mathit{Occ}(\cong_i)(q')$. The result then follows from $(\rho, j) \in \mathit{Occ}(\cong_i)(q')$, $(\rho_1, j) \in \mathit{Occ}(\cong_{i+1})(q')$, $S_i \in \rho$, and $B_i \notin \rho_1$. □

Lemma 6 *There is a $t \leq n-1$ such that $\simeq_t = \cong_t$.*

Proof. As long as the algorithm has not terminated, we have $B_i \subset S_i$ and consequently $\cong_{i+1} \subset \cong_i$. By finiteness of Q it follows that after $t = |Q| - 1$ (if the iterations have not yet terminated) then $\cong_t$ will be equal to $Q \times Q$. The result follows immediately from Lemma 1. $\square$

Now, we are ready to prove correctness. Lemma 6 guarantees that the algorithm terminates, producing $\simeq_t$. According to Lemma 3, $\simeq_t$ is stable with respect to $\cong_t$, and since $\simeq_t = \cong_t$, the equivalence $\simeq_t$ is stable. The implication of this, in combination with Lemma 5, is stated as the following theorem.

Theorem 1 *The algorithm terminates with output $\simeq_t$, where $\simeq_t$ is the coarsest stable refinement of $\simeq_{init}$.*

To simplify the discussion regarding time complexity, we formulate Lemma 7.

Lemma 7 *For each $q \in Q$ and $i < j$, it is the case that if $q \in B_i \cap B_j$ then $|B_j| \leq \frac{|B_i|}{2}$.*

Proof. By definition we know that B_i is a block of $\cong_{i+1}$. Since $i < j$ it follows by definition that $\cong_j$ is a refinement of $\cong_i$ and hence B_i is a union of blocks in $\cong_j$. From the fact that $q \in B_j$ we know that $q \in S_j$. Since $q \in B_i$ it follows that $S_j \subseteq B_i$. From $|B_j| \leq \frac{|S_j|}{2}$, it follows that $|B_j| \leq \frac{|B_i|}{2}$. $\square$

As demonstrated in Section 4, calculation steps 1 to 8 can each be performed in time $O\big(\sum_{r\in\delta_B} |r|\big)$. This is also the time required by an entire iteration. The time complexity of the algorithm can then be written

$$\sum_{r\in\delta_{B_0}} |r| + \sum_{r\in\delta_{B_1}} |r| + \ldots \sum_{r\in\delta_{B_t}} |r| \quad ,$$

where B_i is the B-block chosen during the ith iteration. Now, a transition rule $r = (q_1, \ldots, q_k) \rightarrow q_{k+1} \in \delta$ is contained in the set δ_{B_i}, $0 \leq i \leq t$, if state q_j is contained in B_i for some $j : 1 \leq j \leq k+1$.

Since no state occurs in more than $\log n$ B-blocks (by Lemma 7), and since r contains at most $|r|$ distinct states, r cannot contribute by more than $|r|^2 \log n$ to the total sum. This implies that the algorithm runs in time $O\Big(\big(\sum_{r\in\delta} |r|^2\big) \log n\Big)$, which is bounded by $O(\hat{r}\, m \log n)$.

4. Iterations

In this section we describe the data structures used in the representation of the equivalences $\simeq_i$ and $\cong_i$ (see Section 3). Also, we use a number of auxiliary data structures which allow efficient implementation of each iteration in the algorithm. Finally, we describe how to implement each iteration.

Each state is represented by a record which we identify with the state itself. We maintain three lists of blocks:

- P corresponds to blocks in $\simeq_i$. Each block is represented by a record which we will identify with the block itself. Each block S in P contains a pointer to a doubly linked list of its elements; and each state points to the block in P

containing it. Each block in P is also equipped with a natural number which indicates its size.

- X corresponds to the blocks in $\cong_i$. Each block is represented by a record which we will henceforth identify with the block itself. A block in the list X is said to be *simple* if it contains a single block of P, and *compound* otherwise. Each block in X contains a pointer to a doubly linked list of the blocks of P contained in it; and each block S in P contains a pointer to the block of X containing it.

- C is a sublist of X containing only the compound blocks in X.

The elements of the lists described above are doubly linked. This allows for deletion of elements in constant time. A transition rule r is represented by a doubly linked list of elements. The j^{th} element of this list (which corresponds to state q) is a record with:

- pointers to the next and previous elements of r (if any).

- pointers to the j^{th} element in the previous and the next rule in $[r]_{\cong_i}$.

- a pointer to the symbolic rule $\rho = [r]_{\cong_i}$.

- pointers c, c_1, and c_2 to three counters containing natural numbers.

Intuitively, given a rule r and a position $j : 1 \leq j \leq |r|$, the pointer c points to $count(j)(\rho)(q)$ where $\rho = [r]_{\cong_i}$. The pointers c_1 and c_2 are temporary variables, used during the iterations, to point to the counters $count(j)(\rho')(q)$ resp. $count(j)\left(\llbracket\rho\rrbracket^B_{\cong_{i+1}}\right)(q)$ respectively, where $\rho' = [r]_{\cong_{i+1}}$. A state has a pointer to the list of rules in which it occurs. A symbolic rule ρ is represented by a record which is pointed to by all instances of ρ.

Initialization In the initial configuration, all transition rules $r \in \delta$ point to (the only) symbolic rule $\rho_0 \in (\delta/\cong_0)$. The pointer c in position j of a transition rule r (which corresponds to a state q) points to the counter $count(j)(\rho_0)(q)$. The list X contains only one block. This block is compound and it is also the only block contained in C.

Below we describe in detail the eight steps which are performed during each iteration of the algorithm. Steps 1–2 take constant time, while steps 3–8 are of time complexity

$$O\left(\sum_{r \in \delta_B} |r|\right)$$

Step 1: Select compound block S. Remove a compound block S from C. Examine the first two blocks in S. Let B be the smaller one. If they are equal in size then B can be arbitrarily chosen to be anyone of them. These blocks correspond to B_i and S_i chosen during the i^{th} iteration (Section 3).

Step 2: Remove B from S. This step is to maintain the invariant that $q \cong_{i+1} q'$ implies that $q \in B$ iff $q' \in B$. Remove B from S and create a new block S' in X. The block S' is simple and contains B as its only block. If S is still compound, put it back into C. Observe that the elements of X will now correspond to the blocks of $\cong_{i+1}$.

Step 3: Calculate new symbolic rules. Each symbolic rule $\rho \in (\delta/\cong_i)$ will potentially give rise to a set of rules in $(\delta/\cong_{i+1})$. These rules are obtained from ρ by replacing occurrences of S in ρ either by B or $S - B$. The new rules are of two types: those which are members of $[\![\rho]\!]^B_{\cong_{i+1}}$ and those which are not. The purpose of Step 3 is to derive the rules in $[\![\rho]\!]^B_{\cong_{i+1}}$, i.e., to generate those members of $(\delta/\cong_{i+1})$ in which B occurs at least once. For this purpose, we build, for each ρ with $[\![\rho]\!]^B_{\cong_{i+1}} \neq \emptyset$, a tree T_ρ which encodes the symbolic rules in $[\![\rho]\!]^B_{\cong_{i+1}}$. A list of existing trees is maintained throughout the current iteration. A rule ρ will maintain a pointer to T_ρ, while T_ρ will maintain a pointer to the list of its leafs.

For a tree T_ρ, the edges of T_ρ are labeled with blocks in X (i.e., blocks in $\cong_{i+1}$). Each path π from the root to a leaf is of length $|\rho|$, and corresponds to one symbolic rule $\rho' \in [\![\rho]\!]_{\cong_{i+1}}$. More precisely, the root-to-leaf concatenation of the labels of edges along π defines the blocks which appear in ρ' from left to right. Thus, the j^{th} edge in π is labeled by $\rho'(j)$, for $i : 1 \leq j \leq |\rho'|$. Furthermore, the leaf at the end of π points to the set of rules which are instances of ρ'. This set of rules is stored as a matrix $M_{\rho'}$, where the rows correspond to rules and the columns correspond to given positions in the rules. Each row k of $M_{\rho'}$ is a linked list corresponding to one rule $r \in [\![\rho]\!]_{\cong_{i+1}}$, where position j contains the state $r(j)$ (that is, $M_{\rho'}[k, j]$ contains $r(j)$). The matrix form is created by also letting elements of different rules in $M_{\rho'}$ to be linked together: position j in each rule has a pointer to position j of the next rule in $M_{\rho'}$ and vice versa (i.e., $M_{\rho'}[k, j]$ and $M_{\rho'}[k + 1, j]$ are linked together).

When T_ρ is completely constructed, each symbolic rule $\rho' \in [\![\rho]\!]^B_{\cong_{i+1}}$ will be represented by a path in T_ρ; and each instance of ρ' will be present as a row in the matrix $M_{\rho'}$ associated with the corresponding leaf.

To construct T_ρ, we go through the elements of B. For each element $q \in B$, we go through the list of rules r with $q \in r$. Recall that q has a pointer to this list. To prevent that a certain rule is considered twice, we mark encountered rules (and unmark them at the end of the step). For a rule r, we find the symbolic rule $\rho = [r]_{\cong_i}$. This can be done since each r has a pointer to ρ, and since the existing symbolic rules still correspond to those in $(\delta/\cong_i)$ (they have yet not been modified to reflect $\cong_{i+1}$). We also find the tree T_ρ by following pointer from ρ to T_ρ. If T_ρ does not exist yet, we create it, add it to the list of currently existing trees, and add a pointer to it from ρ. We modify T_ρ by "adding" r to it. The addition process is carried out as follows. Let r be of the form $(q_1, \ldots, q_n) \to q_{n+1}$. We simultaneously traverse r (from left to right) and T_ρ (in a top-down manner). We start from q_1 and the root of the tree. At step j of the traversal, we consider the state q_j together with a node n_j in T_ρ. We check whether there is an edge leaving n_j which is labeled by $[q_j]_{\cong_{i+1}}$ (we can find $[q_j]_{\cong_{i+1}}$ by following the pointer to the block in P containing q_j and from there following the pointer to the corresponding block in X). If such an edge exists, we follow the edge one step down the tree to the next node n_{j+1}. We also move one step to the right in r to the state q_{j+1}. If no such an edge exists, we create a new edge n_{j+1} connected to n_j and labeled with $[q_i]_{\cong_{i+1}}$ (again moving one step to the right in r). Checking existence of the right edge takes constant time. This is due to the fact that each node may have at most two outgoing edges (in fact a node has only outgoing edge unless the edges are labeled by B or S). Once we reach a leaf (after $|\rho|$ steps), we insert r in the matrix pointed to by the leaf. More precisely, we go through r from left to right. For element j in r, we remove any existing (old) links to and from elements of other lists, and add a double link to element j of the rule which was previously (before the insertion of r) the first row in the matrix. This is to maintain the matrix form, i.e., the invariant that corresponding elements in rules in the same matrix are linked. If the leaf has just been created, we add it to the list of leafs of the tree. Notice that if the leaf is new, then its matrix will only contain one row, namely the one corresponding to r.

Step 4: Create counters. In this step, we create new counters to reflect the introduction of the new symbolic rules, and update the values of the temporary pointers c_1 and c_2 in the different positions of the relevant rules. We go through the list of existing trees one after each other. The scanning of a tree T_ρ is carried out in a column-wise fashion. Let $j : 1 \leq j \leq |\rho|$ be the column to be considered next. We go through all leaves of T_ρ, and for each leaf we go through column j of the corresponding matrix. To find column j in each leaf, a leaf in T_ρ (corresponding to a symbolic rule ρ') is equipped with a pointer $\vec{\rho'}$. The pointer $\vec{\rho'}$ points to the position $M_{\rho'}[1,j]$, i.e., position j of the first row in $M_{\rho'}$. At the start of the scanning of T_ρ, the pointer $\vec{\rho'}$ points to $M_{\rho'}[1,1]$. For a given leaf (corresponding to a symbolic rule ρ'), we scan position j of all the rules in $M_{\rho'}$ one by one. This can be done due to the matrix form, where position j in each rule has a pointer to position j of the next rule in $M_{\rho'}$. When column j of $M_{\rho'}$ has been completely scanned, pointer c_1 in $M_{\rho'}[i,j]$ (containing a state q) will point to the counter $\mathit{count}(j)(\rho')(q)$. When column j of all the leaves in T_ρ have been completely scanned, pointer c_2 in $M_{\rho'}[i,j]$ will point to the counter $\mathit{count}(j)\left(\llbracket\rho\rrbracket^{B}_{\cong_{i+1}}\right)(q)$.

Now, we describe how we modify the counters. Suppose that we are currently scanning position $M_{\rho'}[i,j]$; and suppose that this position contains state q. If it is the first time we encounter q during the scanning of column j in $M_{\rho'}$, we create the counter $\mathit{count}(j)(\rho')(q)$, initialize its value to one, and make both q and pointer c_1 of $M_{\rho'}[i,j]$ point to it. If it is not the first time, we find $\mathit{count}(j)(\rho')(q)$ by following the pointer from $M_{\rho'}[i,j]$ to q, and from q to the counter. We increase its value and create a pointer to it from c_1 of $M_{\rho'}[i,j]$. We create and modify $\mathit{count}(j)\left(\llbracket\rho'\rrbracket^{B}_{\cong_{i+1}}\right)(q)$ in a similar manner, with two differences, namely (i) we use c_2 instead of c_1; and (ii) we check whether it is the first time we encounter q during the scanning of column j in the current tree (rather than the current leaf). When the scanning of the j^{th} column of $M_{\rho'}$ has been completed, we move $\vec{\rho'}$ one step to the right. Then, we go through the states of B, and for each $q \in B$, we remove the pointer to $\mathit{count}(j)(\rho')(q)$. When the scanning of the j^{th} column of T_ρ has been completed, we go through the states of B, and for each $q \in B$, we remove the pointer to $count(j)\left(\llbracket\rho'\rrbracket^{B}_{\cong_{i+1}}\right)(q)$.

Step 5: Refine P with respect to B. This step preserves the invariant that $q \simeq_{i+1} q'$ implies $Occ(\cong_{i+1})(B)(q) = Occ(\cong_{i+1})(B)(q')$. Each position $j : 1 \leq j \leq |\rho'|$ may potentially give raise to a split of the blocks in P. A state q_1 which occurs in position j in a rule $r \in \llbracket\rho'\rrbracket$ (i.e., $r(j) = q_1$ for some $j : 1 \leq j \leq |\rho'|$) should not be in the same block as a state q_2 which does not occur in position j of any rule in $\llbracket\rho'\rrbracket$. The reason is that this would imply $Occ(\cong_{i+1})(B)(q) \neq Occ(\cong_{i+1})(B)(q')$. To reflect this in our blocks, we go through all trees and all leafs in a tree. For a leaf corresponding to a symbolic rule ρ', we iterate over all positions $j : 1 \leq j \leq |\rho'|$, and scan position j of all the rules in $M_{\rho'}$ one by one. Let q be the state in the position and rule currently under consideration. We find the block D of P containing q. We create an associated block D' if one does not already exist. We move q to D' decreasing the size of D and increasing the size of D'. During the scanning, we construct a list which contains all blocks which have been split. After we have scanned position j of all rules in $M_{\rho'}$, we go through the new list of blocks. For each block D (and associated block D'), we remove the record for D if it has become empty (all its elements have been moved to D'); otherwise if the block of X containing D has become compound by the split, we add this block to C.

Step 6: Refine P with respect to $S - B$. This step maintains the invariant that $q \simeq_{i+1} q'$ implies that $count(j)(\rho)(q) = count(j)\left(\llbracket\rho\rrbracket^{B_i}_{\cong_{i+1}}\right)(q)$ iff $count(j)(\rho)(q') = count(j)\left(\llbracket\rho\rrbracket^{B_i}_{\cong_{i+1}}\right)(q')$. For each tree T_ρ, and each leaf of T_ρ (corresponding to symbolic rule ρ'), we go through the list $M_{\rho'}$, and scan every rule r in $M_{\rho'}$. Let q be the state of r currently scanned. We determine whether the counters pointed to by c and c_1 have the same values. This corresponds to checking whether $count(j)(\rho)(q) = count(j)\left(\llbracket\rho\rrbracket^{B}_{\cong_{i+1}}\right)(q)$. If the equality holds, we find the block D of P containing q, and create an associated block D' if one does not already exist. Afterwards, the new list of blocks is processed in the same way as in Step 4.

Step 7: Update the counters. This step updates the counters for every state in every rule $\rho' \in \llbracket\rho\rrbracket_{\cong_{i+1}}$ such that $B \notin \rho'$. For each tree T_ρ in the list of trees created in Step 3, we go through all the leaves of T_ρ. For a given leaf and an associated list $M_{\rho'}$, we scan $M_{\rho'}$ column-wise. Let j and q be the column resp. state that is currently scanned. We subtract the value of the counter pointed to by c_2 from that pointed to by c and put the value back in the latter. This corresponds to the assignment $count(j)(\rho)(q) := count(j)(\rho)(q) - count(j)(\llbracket\rho\rrbracket^{B}_{\cong_{i+1}})(q)$. If the subtraction result is equal to zero, we destroy the counter to which c points. To prevent that the same state is processed more than once during the scanning of T_ρ we mark encountered states. When the scanning of all leafs of T_ρ has been completed, we scan all leaves one more time and unmark all states. During the same scan we change the pointer c of a cell and make it point to the same counter as c_2. Now, we destroy, for each state q, the pointers c_1 and c_2.

Step 8: Update symbolic rules. We go through each tree T_ρ. For each leaf we create a new symbolic rule ρ'. We go through the associated list of rules, and make the rules point to ρ' and delete the the pointer from the old symbolic rule. If a symbolic rule becomes empty, we destroy its record. After T_ρ has been processed, it is destroyed.

5. NTA Minimization With Respect To Bisimulation

We now discuss how the algorithm presented in Section 3 can be applied to the minimization of non-deterministic tree automata, with respect to bisimulation. We begin with a formal definition of bisimulation equivalence. Let $A = (Q, \Sigma, \delta, F)$ and $A' = (Q', \Sigma, \delta', F')$ be two NTA. A relation $\simeq \subseteq Q \times Q'$ is a bisimulation relation if the following two conditions hold for all states $q \in Q$ and $q' \in Q'$ such that $q \simeq q'$.

(i) $q \in F$ if and only if $q' \in F'$.

(ii) The fact that $(q_1, \ldots, q_{i-1}, q, q_i, \ldots, q_{k-1}) \to q_k \in \delta$, where $i \leq k$, implies that there exists a rule $\left(q'_1, \ldots, q'_{i-1}, q', q'_i, \ldots, q'_{k-1}\right) \to q'_k \in \delta'$, such that $q_j \simeq q'_j$ for all $j \in \{1, \ldots, k\}$, and vice versa.

States q and q' as above are said to be *bisimilar* (with respect to $\simeq$). We consider A and A' to be *bisimulation equivalent* (and write $A \sim A'$) if there is a bisimulation relation such that every state in Q is bisimilar to a state in Q', and every state in Q' to a state in Q.

Here, a brief remark is in place: When the notion of bisimulation equivalence is extended to allow alphabets containing more than one symbol of a given rank, it is required that the same symbol occurs on both sides of the implication. Note also that if A and A' are bisimulation equivalent NTAs, and the relation between their states is one-to-one, then A and A' are isomorphic.

To produce the unique minimal tree automaton (up to isomorphism) that is bisimilar to a given tree automaton $A = (Q, \Sigma, \delta, F)$, we apply the algorithm of Section 3 to find an equivalence relation $\simeq$ on Q, such that $Q/\simeq$ is the coarsest stable partition of Q, and then output $A_\simeq = (Q/\simeq, \Sigma, \delta/\simeq, F/\simeq)$.

Now, we want to prove that bisimulation equivalence implies language equivalence.

Lemma 8 *Let A and A' be a pair of bisimulation equivalent NTA with state space Q and Q', respectively. If states $q \in Q$ and $q' \in Q'$ are bisimilar, then the equality $L_q(A) = L_{q'}(A')$ holds.*

Proof. The proof is by induction, and since a bisimulation relation is symmetric, it suffices to show one inclusion, as the other follows by symmetry. In the following, the NTA $A = (Q, \Sigma, \delta, F)$ and $A' = (Q', \Sigma, \delta', F')$ are bisimilar with respect to the relation $\simeq$. Assume that $q \in Q$ and $q' \in Q'$ are such that $q \simeq q'$, and that $t \in \mathrm{T}_\Sigma$ is a tree in $L_q(A)$. If the height of t is 0, i.e. if $t = a$ for some $a \in \Sigma^{(0)}$, then $a() \to q \in \delta$, and since q and q' are bisimilar, $a() \to q' \in \delta'$ and $t \in L_{q'}(A')$. Suppose then that Lemma 8 holds for all trees of height i or less, and that $t = f^{(k)}[t_1, \ldots, t_k] \in L_q(A)$ is a tree of height $i+1$. Since $t \in L_q(A)$, there are states $q_1, \ldots, q_k \in Q$ such that $t_i \in L_{q_i}(A)$, for all $i \in [k]$, and the rule

$f(q_1, \ldots, q_k) \to q$ is in δ, and since A and A' are bisimulation equivalent, there is a rule $f(q'_1, \ldots, q'_k) \to q'$ in δ', such that $q_i \simeq q'_i$, for all $i \in [k]$. By the induction hypothesis it follows that $t_i \in L_{q'_i}(A')$, for all $i \in [k]$, and consequently the tree t is in $L_{q'}(A')$. □

Theorem 2 *Bisimulation equivalence implies language equality.*

Proof. Let $A = (Q, \Sigma, \delta, F)$ and $A' = (Q', \Sigma, \delta', F')$ be a pair of bisimulation equivalent NTA. Given that t is in $L(A)$, there is a state $q \in \delta(t) \cap F$, because this intersection cannot be empty. Since A and A' are bisimulation equivalent, there is a state $q' \in F'$ that is bisimilar to q.

According to Lemma 8, $L_q(A) = L_{q'}(A')$ and consequently $t \in L(A')$. The same argument also holds for the opposite direction. □

In the derivation of Theorem 3, which is a non-deterministic version of a result in [6], we make use of two lemmas. To save space, the proofs have been omitted, but the interested reader will find these in [1].

Lemma 9 *The input automaton and the output automaton returned by the minimization algorithm are bisimulation equivalent.*

Theorem 3 *Given an automaton A, the minimization algorithm returns the unique minimal bisimulation-equivalent automaton recognizing $L(A)$.*

Proof. Let $A = (Q, \Sigma, \delta, F)$ be an NTA, and $A_\simeq = (Q/\simeq, \Sigma, \delta/\simeq, F/\simeq)$ the NTA returned by the minimization algorithm. According to Theorem 1, $Q/\simeq$ is the coarsest stable refinement of $\simeq_{init}$. By Lemma 9, automata $A_\simeq$ and A are bisimulation equivalent.

Let $A' = (Q', \Sigma, \delta', F')$ be a minimal NTA bisimulation equivalent with A. Since A and A' are bisimulation equivalent, there is an equivalence relation $\simeq'$ on Q, such that $q \simeq' q'$ if q and q' are both bisimilar to the same state in Q'. The partition $Q/\simeq'$ is stable, and a refinement of $\simeq_{init}$. In combination with the assumption that A' is minimal, we have that $Q/\simeq'$ is the unique coarsest stable refinement of $\simeq_{init}$, and hence that $\simeq = \simeq'$. Since both $A_\simeq$ and A' are bisimulation equivalent to A, they are also bisimulation equivalent to each other and since they each have $|Q/\simeq|$ states, this relation is one-to-one. Hence, $A_\simeq$ and A' are isomorphic. □

6. Experiments

As mentioned in the introduction, we have implemented our algorithm in Java. To test the algorithm on real life examples we used tree automata that arose during computations in the framework of tree regular model checking. Tree regular model checking is the name of a family of techniques for analyzing infinite state systems in which states are represented by trees, set of states by tree automata, and transitions by tree transducers. Most tree regular model checking algorithms rely heavily on efficient methods for checking bisimulation since the automata often increase in size during the verification process and some computations are simply not feasible without minimization.

The NTAs that we have minimized arose during verification of the *Percolate* protocol and the *Leader Election* protocol. The Percolate protocol operates on a tree of processes. The protocol simulates the way results propagate in a set of logical-or gates organized in a tree. A more detailed description on the protocol can be found in [12]. The Leader Election protocol consists of a set of processes (represented by the leaves of a tree) that wants to elect a leader. Each process first decides whether to be a candidate or not. The election process then proceeds in two phases. First, the internal nodes checks their children to see if at least one of them has decided to be a candidate. If that is the case, the internal node becomes a candidate as well. Secondly, the root elects one candidate non-deterministically among its children. After this, every internal node that has been elected, elects one of its children (that has declared that it is a candidate). The protocol is further described in [4]. Table 1 shows the execution time, and the size of the tree automata before and after running our minimization algorithm.

Protocol	Input		Output		Time (s)
	States	Trans.	States	Trans.	
Percolate	18	333	5	38	0.2
	21	594	5	45	1.3
Leader	25	384	9	43	0.3
	49	3081	14	167	30.6

Table 1: The results from applying the bisimulation minimization algorithm to tree automata that arose in the verification of protocols Perculate and Leader.

7. Conclusion and Future Work

We have extended an algorithm by Paige and Tarjan for solving the coarsest stable partition problem to the domain of trees, and obtained a running time of $O(\hat{r}\, m \log n)$, where $\hat{r}$ is the maximum rank of the input alphabet, m is the total size of the transition table, and n is the number of states. As demonstrated, the extended algorithm can be used to minimize non-deterministic tree automata with respect to bisimulation equivalence.

One direction for future work is to integrate the minimization algorithm in the framework of tree regular model checking, where tree automata are encoded symbolically. Since many of the algorithms in this framework rely heavily on minimization, we believe that the performance would be improved if our algorithm could be integrated in this setting. We plan to implement a symbolic version of our algorithm where we consider both binary decision diagrams and SAT solvers to perform the necessary operations on the symbolic encoding.

Another direction for future work is to extend our algorithm to work on non-deterministic automata on unranked trees. Unranked tree automata are used in numerous areas of XML-related research [16]. For example, in the context of XML schema languages a minimized schema would improve the running time or memory consumption for document validation.

Unranked tree automata can be modeled in different ways. In [9], a minimization algorithm for deterministic unranked tree automata is described, while complexity results for a different model of deterministic unranked tree automata can be found in [14].

Acknowledgements

We would like to thank Andreas Maletti and Frank Drewes for useful comments.

References

1. P.A. Abdulla and J. Högberg and L.Kaati, "Bisimulation Minimization Of Tree Automata" Technical report, Uppsala University, 2006.
2. P.A. Abdulla and B. Jonsson and P.Mahata and J. d'Orso, "Regular Tree Model Checking", in *CAV, LNCS 2404*, 2002.
3. P. A. Abdulla and B.Jonsson and M.Nilsson and M.Saksena, "A Survey of Regular Model Checking", in *CONCUR, LNCS 3170*, 2004.
4. P.A. Abdulla and A.Legay and J.d'Orso and A.Rezine, "Tree Regular Model Checking: A Simulation-Based Approach", To appear in *Jour. of Logic and Alg. Programming*, 2006.
5. M.Biehl and N.Klarlund and T.Rauhe, "Algorithms for guided tree automata", in *Proceedings of the First Workshop on Implementing Automata, LNCS 1260*, 1997.
6. W. S. Brainerd, "The Minimalization of Tree Automata", in *Information and Computation*, 1968.
7. H. Comon and M. Dauchet and R. Gilleron and F. Jacquemard and D. Lugiez and S. Tison and M. Tommasi, "Tree Automata Techniques and Applications", Available on: http://www.grappa.univ-lille3.fr/tata, 1997.
8. J. Carme and R. Gilleron and A. Lemay and A. Terlutte and M. Tommasi, "Residual finite tree automata", Technical report, GRAPPA, 2003.
9. J.Cristau and C. Löding and W.Thomas, "Deterministic automata on unranked trees", in *FCT, LNCS 3623*, 2005.
10. G.Gramlich and G.Schnitger, "Minimizing NFA's and Regular Expressions", in *STACS, LNCS 3404*, 2005.
11. J. E. Hopcroft, "An *n log n* algorithm for minimizing states in a finite automaton", in *Theory of Machines and Computations*, 1971.
12. Y. Kesten and O. Maler and M. Marcus and A. Pnueli and E. Shahar, "Symbolic Model Checking with Rich Assertional Languages", in *TCS*, volume 256, 2001.
13. R. Milner, "A Calculus of Communicating Systems", in *LNCS 92*, 1980.
14. W. Martens and J. Niehren, "On the Minimization of XML Schemas and Tree Automata for Unranked Trees", in *JCSS*, 2006.
15. A. R. Meyer and L. J. Stockmeyer, "The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space", in *Proc. 13th Ann. IEEE Symp. on Switching and Automata Theory*, 1972.
16. F. Neven, "Automata, Logic, and XML", in *CSL*, 2002.
17. M. Nivat and A. Podelski, "Minimal Ascending and Descending Tree Automata", in *SIAM Journal on Computing (26)*, 1997.
18. R. Paige and R. Tarjan, "Three partition refinement algorithms", in *SIAM Journal on Computing (16)*, 1987.