

# Bisimulation Minimization of Tree Automata

Parosh Aziz Abdulla<sup>1</sup>, Lisa Kaati<sup>1</sup>, and Johanna Högberg<sup>2</sup>

<sup>1</sup> Dept. of Information Technology, Uppsala University, Sweden

{parosh, lisa.kaati}@it.uu.se

<sup>2</sup> Dept. of Computing Science, Umeå University, Sweden

johanna@cs.umu.se

**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.

## 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 the algorithm proposed in [13] 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 [13] 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 [9], 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 (NFA) is PSPACE complete [10] and, what is worse, that the minimization problem for an NFA with  $n$  states cannot be efficiently approximated within the factor  $O(n)$ , unless  $P = PSPACE$  [8]. To avoid exponential time, the problem must either be restricted (i.e. 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.

Bisimulation minimization of tree automata is of particular interest in *tree regular model checking* (an extension of regular model checking). 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 non-deterministic tree automata [2,3].

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). 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 compared to  $O(m \log n)$  by [13]. Thus, interestingly, the maximum rank  $\hat{r}$  (which is the constant 1 in [13]) does not become an exponent. Instead, it influences the complexity rather modestly.

**Related work.** There does not seem to be any 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  $O(nm)$ , but it is an open question whether or not tree automata can be minimized in time  $O(m \log m)$ . In [7], Cristau et al. claim that a deterministic bottom-up tree automaton for unranked trees can be minimized in time  $O(m^2)$ , using the algorithm proposed in [9]. In [6], the minimal tree automata is computed using an algorithm that construct congruences on the states of a given deterministic tree automata until a fixed point is reached. However, there are no given results regarding the complexity of the given algorithm. An alternative definition of deterministic top-down tree automata together with a minimization algorithm is given by Nivat and Podelski in [12].

**Outline.** Section 2 covers the preliminaries, while Section 3 generalizes a partitioning algorithm from [13] 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  $T_\Sigma$  of all *trees over*  $\Sigma$  is the smallest superset of  $\Sigma_{(0)}$  that contains every  $f[t_1, \dots, t_k]$ , where  $f \in \Sigma_{(k)}$ ,  $k \geq 1$ , and  $t_1, \dots, t_k \in T_\Sigma$ . A subset of  $T_\Sigma$  is called a *tree language*.

A *non-deterministic tree automaton* is a quadruple  $A = (Q, \Sigma, \delta, F)$  where  $\Sigma$  is a ranked input alphabet,  $Q$  is a finite set of *states*,  $\delta$  is a finite set of *transition rules*  $f(q_1, \dots, q_n) \rightarrow q_{n+1}$  such that  $f \in \Sigma_{(n)}$ , and  $q_1, \dots, q_{n+1} \in Q$ , for some  $n \in \mathbb{N}$ . Finally,  $F \subseteq Q$  is a set of *accepting* states. In the obvious way,  $\delta$  extends to trees, yielding a relation  $\delta: T_\Sigma \rightarrow \mathcal{P}(Q)$ : For  $t = f[t_1, \dots, t_k] \in T_\Sigma$ ,

$$\delta(t) = \{q \mid f(q_1, \dots, q_k) \rightarrow q \in \delta \text{ and } q_i \in \delta(t_i) \text{ for all } i \in \{1, \dots, k\}\}.$$

The tree language *recognized* by  $A$  is  $L(A) = \{t \in T_\Sigma \mid \delta(t) \cap F \neq \emptyset\}$ . Let  $r = f(q_1, \dots, q_n) \rightarrow q_{n+1}$  be a transition rule, then  $|r|$  denotes its length (that is,  $|r| = n + 1$ ),  $r(i)$  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  $\delta_B$  as the set  $\{r \in \delta \mid \exists q \in B \text{ s.t. } q \in r\}$ .

For technical convenience, we shall henceforth restrict ourselves to ranked alphabets containing at most one symbol of rank  $k$  for each  $k \in \mathbb{N}$ . We can thus leave out the input symbol when writing a transition rule, without risk of confusion. Extending the algorithm presented in Section 3 to unrestricted alphabets is straight-forward and does not effect the results in any way.

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

**Symbolic rules.** Let  $A = (Q, \Sigma, \delta, F)$  be an NTA, and  $\simeq', \simeq$ , where  $\simeq' \subseteq \simeq$ , equivalence relations on  $Q$ . To represent the set  $\{([q_1]_\simeq, \dots, [q_k]_\simeq) \rightarrow [q]_\simeq \mid (q_1, \dots, q_k) \rightarrow q \in \delta\}$  of *symbolic rules*, we use the notation  $(\delta/\simeq)$ . Conversely, if  $\rho = (D_1, \dots, D_k) \rightarrow D_{k+1}$  is a symbolic rule, then the set of *instances* of  $\rho$ , denoted  $\llbracket \rho \rrbracket$ , is the set  $\{(q_1, \dots, q_k) \rightarrow q_{k+1} \mid q_i \in D_i, i : 1 \leq i \leq k + 1\} \cap \delta$ . We write  $\rho(i)$  to refer to block  $D_i$  of  $\rho$ , and  $B \in \rho$  to indicate that  $\rho(i) = B$ , for some  $i \in \{1, \dots, |\rho|\}$ . The length of  $\rho$  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  $r \in \llbracket \rho \rrbracket$ .

Given a rule  $\rho = (D_1, \dots, D_n) \rightarrow D_{n+1}$  in  $(\delta/\simeq)$ , we let  $\llbracket \rho \rrbracket_{\simeq'}$  represent the set

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

To denote the subset  $\{(D'_1, \dots, D'_n) \rightarrow D'_{n+1} \mid \exists i \in \{1, \dots, n + 1\} \text{ s.t. } D'_i = B\}$  of  $\llbracket \rho \rrbracket_{\simeq'}$ , we write  $\llbracket \rho \rrbracket_{\simeq'}^B$ . 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  $\rho' \in \llbracket \rho \rrbracket_{\simeq'}$ .

**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  $(\rho, i)$  where  $\rho \in (\delta/\simeq)$  and  $q \in \rho(i)$  for some  $i : 1 \leq i < |\rho|$ . Intuitively,  $Occ(\simeq)(q)$  identifies the symbolic rules in which  $[q]$  occurs in the left hand side of 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  $\rho$ , and a state  $q$ , we define  $count(\rho)(q)$  to be the size of the set  $\{r \in \llbracket \rho \rrbracket \mid \exists i \in \{1 \dots |\rho|\} \text{ s.t. } r(i) = q\}$ . We extend the definition to a set  $\varrho$  of symbolic rules such that  $count(\varrho)(q) = \sum_{\rho \in \varrho} count(\rho)(q)$ .

**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 equivalence relation  $\simeq_{init}$  is assumed to satisfy the following conditions: (i)  $\simeq_{init}$  is stable with respect to  $Q \times Q$  and (ii) if  $q \simeq_{init} q'$  then  $q \in F$  iff  $q' \in F$ . 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) generating two sequences of equivalence relations on  $Q$ , denoted by  $\simeq_0, \simeq_1, \dots, \simeq_t$  and  $\cong_0, \cong_1, \dots, \cong_t$  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,  $q \simeq_i q'$ , 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:

- (1)  $q \simeq_i q'$
- (2)  $Occ(\cong_{i+1})(B_i)(q) = Occ(\cong_{i+1})(B_i)(q')$
- (3) For every  $\rho \in (\delta/\cong_i)$ , we have that

$$count(\rho)(q) = count\left(\llbracket \rho \rrbracket_{\cong_{i+1}}^{B_i}\right)(q) \quad \text{iff} \quad count(\rho)(q') = count\left(\llbracket \rho \rrbracket_{\cong_{i+1}}^{B_i}\right)(q') .$$

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  $q \simeq_i q$  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  $q \cong_{i+1} q'$ .  $\square$

This implies that  $\simeq_i$  is a *proper* refinement 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 2 and Lemma 5.

**Lemma 2.** *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  $\simeq_0$ . Suppose then that  $q \simeq_{i+1} q'$ , and that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(q)$  for some  $\rho \in (\delta/\simeq_{i+1})$ ; we show that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(q')$ . Depending on  $\rho$ , we have three cases:

First,  $S_i - B_i \notin \rho$  and  $B_i \notin \rho$ . This means that  $\rho \in (\delta/\simeq_i)$ , and therefore  $(\rho, j) \in \text{Occ}(\simeq_i)(q)$ . 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  $\simeq_i$ , and hence  $(\rho, j) \in \text{Occ}(\simeq_i)(q')$ . Since  $\rho \in (\delta/\simeq_{i+1})$  it follows that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(q')$ .

Second,  $S_i - B_i \in \rho$  and  $B_i \notin \rho$ . Let  $\rho$  be of the form  $(D_1, \dots, D_n) \rightarrow D_{n+1}$ . Define  $\rho^1 \in (\delta/\simeq_i)$  to be the symbolic rule  $(D_1^1, \dots, D_n^1) \rightarrow D_{n+1}^1$  where, for each  $k : 1 \leq k \leq n+1$ , we have that  $D_k^1 = S_i$  if  $D_k^1 = S_i - B_i$  and  $D_k^1 = D_k$  otherwise. We observe that  $\rho^1 = [\rho]_{\simeq_i}$ , and therefore  $(\rho^1, j) \in \text{Occ}(\simeq_i)(q)$ . 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  $\simeq_i$ , and hence  $(\rho^1, j) \in \text{Occ}(\simeq_i)(q')$ . From  $B_i \notin \rho$  we know that  $\text{count}(\rho)(q) > \text{count}([\rho]_{\simeq_{i+1}}^{B_i})(q)$ . Since  $q \simeq_{i+1} q'$  it follows that  $\text{count}(\rho)(q') > \text{count}([\rho]_{\simeq_{i+1}}^{B_i})(q')$ . Hence,  $(\rho, j) \in \text{Occ}(\simeq_i)(q')$ .

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

In the proof of Lemma 5, we use two auxiliary lemmas (Lemma 3 and Lemma 4). The proofs of these two lemmas have been omitted, but the interested reader will find these in [4].

**Lemma 3.** *Any stable refinement  $\simeq$  of  $\simeq_i$ , is also a stable refinement of  $\simeq_{i+1}$ .*

**Lemma 4.** *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  $(\rho, j) \in \text{Occ}(\simeq)(q)$  if and only if  $(\rho', j) \in \text{Occ}(\simeq')(q)$  for some  $\rho' \in [\rho]_{\simeq'}$ .*

**Lemma 5.** *If  $\simeq$  is a stable refinement of  $\simeq_0$ , then  $\simeq$  is also a 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  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(B_i)(q)$ . Since  $(\rho, j)$  is in  $\text{Occ}(\simeq_{i+1})(B_i)(q)$  we know that  $B_i \in \rho$  and that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(q)$ . From the induction hypothesis we know that  $\simeq \subseteq \simeq_i$ , and by Lemma 3 that  $\simeq \subseteq \simeq_{i+1}$ . By Lemma 4 there is a  $\rho' \in [\rho]_{\simeq}$  such that  $(\rho', j) \in \text{Occ}(\simeq)(q)$ . Since  $q \simeq q'$  and  $\simeq$  is stable, we have that  $(\rho', j) \in \text{Occ}(\simeq)(q')$ . From  $\simeq \subseteq \simeq_{i+1}$  and  $\rho' \in [\rho]_{\simeq}$ , it follows by Lemma 4 that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(q')$ . Since  $B_i \in \rho$ , we conclude that  $(\rho, j) \in \text{Occ}(\simeq_{i+1})(B_i)(q')$ .

Regarding Condition (3), assume that  $\text{count}(\rho)(q) \neq \text{count}([\rho]_{\simeq_{i+1}}^{B_i})(q)$ . We show that  $\text{count}(\rho)(q') \neq \text{count}([\rho]_{\simeq_{i+1}}^{B_i})(q')$ . From the above assumption, we know that there are  $\rho_1 \in (\rho/\simeq_i)$  and  $j$  such that  $B_i \notin \rho_1$  and  $(\rho_1, j)$  is an element of  $\text{Occ}(\simeq_{i+1})(q)$ . From the induction hypothesis we know that  $\simeq \subseteq \simeq_i$ ,

and hence by Lemma 3 it follows that  $\simeq \subseteq \cong_{i+1}$ . By Lemma 4 there is a  $\rho_2 \in \llbracket \rho_1 \rrbracket_{\simeq}$ , such that  $(\rho_2, j) \in \text{Occ}(\simeq)(q)$ . Since  $q \simeq q'$  and  $\simeq$  is stable it follows that  $(\rho_2, j) \in \text{Occ}(\simeq)(q')$ . From  $\simeq \subseteq \cong_{i+1}$  and  $\rho_2 \in \llbracket \rho_1 \rrbracket_{\simeq}$ , it follows by Lemma 4 that  $(\rho_1, j)$  is an element of  $\text{Occ}(\cong_{i+1})(q')$ , and hence the result.  $\square$

**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 at most  $t = |Q| - 1$  steps we reach a point where there are no  $B_t \in (Q/\simeq_t)$  and  $S_t \in (Q/\simeq_t)$  such that  $B_t \subset S_t$  and  $|B_t| \leq \frac{|S_t|}{2}$ . This implies  $\simeq_t = \cong_t$ .  $\square$

Now, we are ready to prove correctness. Lemma 6 guarantees that the algorithm terminates, producing  $\simeq_t$ . According to Lemma 2,  $\simeq_t$  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  $\simeq_0$ .*

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

**Lemma 7.** *For each  $q \in Q$  and  $i < j$  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  $O(\sum_{r \in \delta_B} |r|)$ . This is also the time required by an entire iteration. The time complexity of the algorithm can then be written as

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

where  $B_i$  is the  $B$ -block chosen during the  $i$ th iteration. Now, a transition rule  $r = (q_1, \dots, q_k) \rightarrow q_{k+1} \in \delta$  will only be contained in the set  $\delta_{B_i}$ ,  $0 \leq i \leq t$ , if state  $q_j$  is contained in  $B_i$  for some  $j : 1 \leq j \leq k + 1$ . No state occur in more than  $\log n$   $B$ -blocks (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  $O\left(\sum_{r \in \delta} |r|^2 \log n\right)$ , 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  $\cong_i$ . 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 identify with the block itself. A block of  $X$  is *simple* if it contains a single block of  $P$ , and is *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 above lists are doubly linked. This allows deletion of elements in constant time. A rule  $r$  is represented by a doubly linked list of elements. The  $i^{th}$  element of the list (corresponding to state  $q$ ) is a record with:

- pointers to the next and previous elements of  $r$  (if any).
- pointers to the  $i^{th}$  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  $r$ , the pointer  $c$  points to  $count(\rho)(q)$  where  $\rho = [r]_{\cong_i}$ . The counters  $c_1$  and  $c_2$  are temporary variables, used during the iterations, to point to  $count(\rho')(q)$  resp.  $count\left(\llbracket \rho \rrbracket_{\cong_{i+1}}^B\right)(q)$ , where  $\rho' = [r]_{\cong_{i+1}}$ . A state has a pointer to the list of rules in which it occurs. A symbolic rule  $\rho$  is represented by a record which is pointed to by all instances of  $\rho$ .

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

**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). This step can be performed in constant time.

**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  $\cong_{i+1}$ . This step can be performed in constant time.

**Step 3: Calculate new symbolic rules.** Note that each symbolic rule  $\rho \in (\delta/\cong_i)$  will potentially give raise to a set of rules in  $(\delta/\cong_{i+1})$ , namely those in  $\llbracket \rho \rrbracket_{\cong_{i+1}}^B$  and  $\llbracket \rho \rrbracket_{\cong_{i+1}}^{\neg B}$ , and that these rules are obtained from  $\rho$  by replacing occurrences of  $S$  in  $\rho$  either by  $B_i$  or  $S - B$ . The purpose of Step 3 is to derive the rules in  $\llbracket \rho \rrbracket_{\cong_{i+1}}^B$ , 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  $\rho$  with  $\llbracket \rho \rrbracket_{\cong_{i+1}}^B \neq \emptyset$ , a tree  $T_\rho$  which encodes the symbolic rules in  $\llbracket \rho \rrbracket_{\cong_{i+1}}^B$ . A list of existing trees is maintained throughout the current iteration. The rule  $\rho$  will maintain a pointer<sup>1</sup> to  $T_\rho$ , while each tree will maintain a pointer to the list of its leafs.

The edges of the tree are labeled with blocks in  $X$  (i.e., blocks in  $\cong_{i+1}$ ). Each path  $\pi$  from the root to a leaf is of length  $|\rho|$ , and corresponds to one symbolic rule  $\rho' = [\rho]_{\cong_{i+1}}$ . More precisely, the root-to-leaf concatenation of the labels of edges along  $\pi$  defines the blocks which appear in  $\rho'$  from left to right. Thus, the  $i^{th}$  edge in  $\pi$  is labeled by  $\rho'(i)$ , for  $i : 1 \leq i \leq |\rho'|$ . Furthermore, the leaf at the end of  $\pi$  points to a list  $L_{\rho'}$  of rules which are instances of  $\rho'$ . The elements of different rules in  $L_{\rho'}$  are also linked together: position  $j$  in each rule has a pointer to position  $j$  of the next rule in  $L_{\rho'}$ . This gives the list  $L_{\rho'}$  a “matrix” form where the rows correspond to rules and the columns correspond to given positions in the rules. When  $T_\rho$  is completely constructed, each symbolic rule  $\rho' \in \llbracket \rho \rrbracket_{\cong_{i+1}}^B$  will be represented by a path in  $T_\rho$ ; and each instance of  $\rho'$  will be present in the list associated with the corresponding leaf.

To construct  $T_\rho$ , 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  $\rho$ , 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  $T_\rho$  by following pointer from each symbolic rule  $\rho$  to  $T_\rho$ . If  $T_\rho$  does not exist yet, we create it, add it to the list of currently existing trees, and add a pointer to it from  $\rho$ . Now we modify  $T_\rho$  by “adding”  $r$  to it. The addition process is carried out as follows. Let  $r$  be of the form  $(q_1, \dots, q_n) \rightarrow q_{n+1}$ . We simultaneously traverse  $r$  (from left to right) and  $T_\rho$  (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  $T_\rho$ . 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  $B$

<sup>1</sup> Pointer from each symbolic rule  $\rho$  to  $T_\rho$ .



or  $S$ ). Once we reach a leaf (after  $|\rho|$  steps), we insert  $r$  in the list 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 first in the list of rules (before the insertion of  $r$ ). This is to maintain the matrix form, i.e., the invariant that corresponding elements in rules in the same list are linked. If the leaf had just been created, we add it to the list of leafs of the tree. Notice that the time complexity of the current step is  $O(\sum_{r \in \delta_B} |r|)$ . In fact, as we shall see all subsequent steps have the same complexity.

**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 relevant rules. We go through the list of existing trees and through the list of leafs of each tree. For a given leaf corresponding to a symbolic rule  $\rho'$ , we consider the corresponding list  $L_{\rho'}$ , and consider each rule  $r$  in the corresponding list. We scan the rule  $r$ , and each position (corresponding to a state  $q$ ). If it is the first time we encounter  $q$  during the scanning of the current leaf, we create the counter  $count(\rho')(q)$ , and make both  $q$  and pointer  $c_1$  of the current position point to it. If it is not the first time, we find  $count(\rho')(q)$  by following the pointer from the current position to  $q$ , and from  $q$  to the counter. We increase its value and create a pointer to it from  $c_1$  of the current position. We create and modify  $count\left(\llbracket \rho \rrbracket_{\cong_{i+1}}^B\right)(q)$  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 the current tree (rather than the current leaf). To prevent that the same is considered twice during the scanning of  $r$ , we mark encountered states. When the scanning of  $r$  has been completed, we scan  $r$  one more time and unmark all states. When we have scanned all rules in the current leaf, we go through all rules and positions one more time and delete the pointers we have created from states  $q$  to the counters  $count(\rho')(q)$  (preserving the ones from  $c_2$ ). When we have scanned all leafs in the current tree, we delete the corresponding pointers to  $count\left(\llbracket \rho \rrbracket_{\cong_{i+1}}^B\right)(q)$ .

**Step 5: Refine  $P$  with respect to  $B$ .** Each position  $j : 1 \leq j \leq |\rho'| - 1$  may potentially give rise to a split of the blocks in  $P$ . A state  $q_1$  which occurs in position  $j$  in the left hand side of a rule  $r \in \llbracket \rho' \rrbracket$  (i.e.,  $r(j) = q_1$  for some  $j : 1 \leq j \leq |\rho'| - 1$ ) 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 rule  $\rho'$ , we iterate over all positions  $j : 1 \leq j \leq |\rho'| - 1$ , and scan position  $j$  of all the rules in  $L_{\rho'}$  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  $L_{\rho'}$ . 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  $L_{\rho'}$ , 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  $S - B$ .** For each tree  $T_\rho$ , and all of its leaves, we go through the list  $L_{\rho'}$ , and scan every rule  $r$  in  $L_{\rho'}$ . 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  $\text{count}(\rho)(q) = \text{count}(\llbracket \rho \rrbracket_{\cong_{i+1}}^B)(q)$ . 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. Afterward, 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 in  $\llbracket \rho \rrbracket_{\cong_{i+1}}^B$ . For each tree  $T_\rho$  in the list of trees created in Step 3, we go through all the leaves of  $T_\rho$ . For a given leaf and an associated list  $L_{\rho'}$ , we scan each rule  $r$  in  $L_{\rho'}$  from left to right. Let  $q$  be the 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  $\text{count}(\rho)(q) := \text{count}(\rho)(q) - \text{count}(\llbracket \rho \rrbracket_{\cong_{i+1}}^B)(q)$ . To prevent that the same state is processed more than once, we mark encountered states. When the scanning of all leaves of  $T_\rho$  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  $c_2$  and the corresponding counters.

**Step 8: Update symbolic rules.** We go through each tree  $T_\rho$ . For each leaf we create a new symbolic rule  $\rho'$ . We go through the associated list of rules, and make the rules point to  $\rho'$ . After  $T_\rho$  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  $q \simeq q'$ . First,  $q \in F$  if and only if  $q' \in F'$ . Second, the fact that  $(q_1, \dots, q_{i-1}, q, q_i, \dots, q_{k-1}) \rightarrow q_k \in \delta$ , where  $i \leq k$ , implies that there exists a rule  $(q'_1, \dots, q'_{i-1}, q', q'_i, \dots, q'_{k-1}) \rightarrow q'_k \in \delta'$ , such that  $q_j \simeq q'_j$  for all  $j \in \{1, \dots, k\}$ , 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, one must require that it is the same symbol that 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.

Now, to produce the unique minimal tree automaton that is bisimilar to a given tree automaton  $A = (Q, \Sigma, \delta, F)$ , we first apply the algorithm of Section 3 with  $\simeq_0 = \simeq_{init}$  to find an equivalence relation  $\simeq$  on  $Q$ , such that  $Q/\simeq$  is the coarsest stable partition of  $Q$ , and then output  $A_\simeq = (Q/\simeq, \Sigma, \delta/\simeq, F/\simeq)$ .

In the derivation of Theorem 2, 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 [4].

**Lemma 8.** *Bisimulation equivalence is an equivalence relation.*

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

**Theorem 2.** *Given an automaton  $A$ , the minimization algorithm returns the unique minimal bisimulation-equivalent automaton recognizing  $L(A)$ .*

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 (Lemma 8), and since they each have  $|Q/\simeq|$  states, this relation is one-to-one. Hence,  $A_\simeq$  and  $A'$  are isomorphic.  $\square$

Note that all deterministic tree automata (DTA) that recognize the same language are bisimulation equivalent to each other, because they are all bisimulation equivalent to the unique minimal DTA that recognizes this language, and bisimulation equivalence, like all equivalence relations, is transitive.

**Corollary 1.** *Given a DTA  $A$ , the minimization algorithm returns the unique minimal DTA recognizing  $L(A)$ .*

The behavior described in Corollary 1 makes it impossible to give a nontrivial approximation bound for the performance of the minimization algorithm. To see why, recall that there is a family of tree languages  $\mathcal{T}$  such that if  $L \in \mathcal{T}$ , then the minimal DTA that recognizes  $L$  is exponentially larger than a minimal

NTA recognizing the same language. This means that given any DTA  $A$  such that  $L(A) \in \mathcal{T}$ , the algorithm will return a potentially smaller, but still deterministic, tree automaton, and hence misses the optimum with an exponential factor.

## 6 Experiments

To test our algorithm on some real life examples, we have used tree automata that arose during computations in the framework of tree regular model checking of some of the protocols described in [1]. Table 1 shows the execution time, and the size before and after running our minimization algorithm.

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.

**Table 1.** Minimization of non-deterministic tree automata

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

## 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 possible 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 it would improve performance 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 possibility is to extend the algorithm to work on hedge automata (HA). As described in [11], an XML document can be viewed as a *hedge* (a more general type of tree), and validated using a HA. We expect the size of this HA to affect the efficiency with which the validation can be performed.

## References

1. P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. Tree regular model checking: A simulation-based approach. *To appear in Jour. of Logic and Alg. Programming*, 2006.
2. P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 555–568. Springer, 2002.
3. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.
4. P. A. Abdulla, L. Kaati, and J. Högborg. Minimization of tree automata. UMINF 06.25, Department of Computer Science, Umeå University, 2006.
5. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *Proceedings of the First Workshop on Implementing Automata*, volume 1260 of *Lecture Notes in Computer Science*. Springer, 1997.
6. W. S. Brainerd. The minimalization of tree automata. *Information and Computation*, 13:484–491, 1968.
7. J. Cristau, C. Löding, and W. Thomas. Deterministic automata on unranked trees. In *FCT*, volume 3623 of *Lecture Notes in Computer Science*. Springer, 2005.
8. G. Gramlich and G. Schnitger. Minimizing nfa's and regular expressions. In *STACS*, volume 3404 of *Lecture Notes in Computer Science*. Springer, 2005.
9. J. E. Hopcroft. An  $n \log n$  algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*. Academic Press, 1971.
10. A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions. In *Proc. 13th Ann. IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.
11. M. Murata. “Hedge Automata: a Formal Model for XML Schemata”. Web page, 2000.
12. M. Nivat and A. Podelski. Minimal ascending and descending tree automata. *SIAM Journal on Computing*, 26:39–58, 1997.
13. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16:973–989, 1987.