On Minimizing Deterministic Tree Automata

Loek Cleophas, Derrick G. Kourie, Tinus Strauss, and Bruce W. Watson

FASTAR Research Group, Department of Computer Science, University of Pretoria,
0002 Pretoria, Republic of South Africa, http://www.fastar.org
loek@loekcleophas.com, dkourie@cs.up.ac.za, tstrauss@cs.up.ac.za, bruce@fastar.org

Abstract. We present two algorithms for minimizing deterministic frontier-to-root tree automata (DFRTAs) and compare them with their string counterparts. The presentation is incremental, starting out from definitions of minimality of automata and state equivalence, in the style of earlier algorithm taxonomies by the authors. The first algorithm is the classical one, initially presented by Brainerd in the 1960s and presented (sometimes imprecisely) in standard texts on tree language theory ever since. The second algorithm is completely new. This algorithm, essentially representing the generalization to ranked trees of the string algorithm presented by Watson and Daciuk, incrementally minimizes a DFRTA. As a result, intermediate results of the algorithm can be used to reduce the initial automaton's size. This makes the algorithm useful in situations where running time is restricted (for example, in real-time applications). We also briefly sketch how a concurrent specification of the algorithm in CSP can be obtained from an existing specification for the DFA case.

Keywords: deterministic frontier-to-root tree automata, deterministic bottom-up tree automata, minimization, minimality

1 Introduction

Minimization of deterministic finite *string* automata (DFAs) has been studied since the late 1950s. Many applications of such minimization arose, and as a result many algorithms were published, often with vastly differing presentation styles and levels of formality [12]. For the case of deterministic frontier-to-root (aka bottom-up) tree automata (DFRTAs), minimization was considered less frequently, likely due to fewer applications being considered at the time. Minimization for DFRTAs was first discussed in the late 1960s by Brainerd [1,2], who presented a textual procedure for minimization that is essentially the generalization to trees of a classical DFA minimization approach. Later standard references either do not discuss minimization at all or present an approach similar to Brainerd's. Later standard references either do not discuss minimization at all [8], have a discussion similar to Brainerd's [9], or give a somewhat imprecise algorithm [6]. As pointed out by Carrasco, Daciuk and Forcada [3], discussions of an implementation of such a minimization algorithm are hard to find. Their paper presents such a discussion for the case of deterministic bottom-up tree automata *over unranked trees*. Carrasco et al. also presented an algorithm for *incremental construction* of minimal deterministic bottom-up tree automata over unranked trees [4].

For the string case, Watson presented an extensive taxonomy of minimization algorithms [12, Chapter 7]. A concurrent specification of an incremental minimization algorithm for the string case was recently presented by Strauss et al. [11], offering possibilities for exploiting parallelism on systems or networks of systems with multiple CPU cores.

Loek Cleophas, Derrick G. Kourie, Tinus Strauss, Bruce W. Watson: On Minimizing Deterministic Tree Automata, pp. 173–182.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

Both in the string and the tree case, minimization is based on the notion of language equivalence between states; either in the form of that equivalence relation, its complement (i.e. the distinguishability relation), or of the state set partition induced by the equivalence relation. In the classical approach, the algorithms start off with a set of possibly equivalent state pairs. This is then refined iteratively by removing state pairs that are definitely not equivalent, until the greatest fixed-point is reached. The resulting set defines a partitioning of the states set into equivalence classes which correspond to the state set of the minimal automaton equivalent to the original one. Put differently, the algorithms compute the greatest fixed point from the top i.e. from the unsafe side [13], meaning that intermediate results of the algorithm cannot be used to reduce the initial DFA.

Watson in [12] and most recently Watson & Daciuk in [13] present an incremental approach to DFA minimization. Their approach results in an algorithm that starts out with a singleton partition for each of the states of the initial DFA and refines this partition by iteratively merging partitions that are shown to be equivalent. The greatest fixed-point reached corresponds to the state set of the minimal automaton equivalent to the original one. Such an algorithm thus computes the greatest fixed point from below i.e. from the safe side. Clearly, intermediate results from such an algorithm can already be used to reduce the original DFA.

In this paper, we focus on minimization of DFRTAs. We present both an algorithm using the classical approach and a new algorithm using the incremental approach to minimization. The latter is the first description of such an algorithm for the tree case. The former is presented more precisely than in most existing literature (with the exception of [3], although that work considers the case of *unranked* deterministic bottom-up tree automata). Furthermore, its inclusion allows one easily to compare and contrast the two approaches (as is the case for DFA minimization algorithms in the taxonomy of such algorithms in [12, Ch. 7]).

We also briefly consider the generalization to the DFRTA case of an existing concurrent specification in CSP of the incremental DFA minimization algorithm. This elegant generalization further increases the parallelism in the specification.

The rest of this paper is structured as follows:

- Section 2 discusses some preliminaries on DFAs, trees, and DFRTAs needed in the remainder of the paper.
- Equivalence of states and minimality of DFAs and DFRTAs are discussed in Section 3.
- Section 4 discusses the classical approach to minimization of DFAs and DFRTAs.
- The incremental approach to minimization and our resulting new incremental minimization algorithm for DFRTAs is discussed in Section 5.
- Section 6 presents a concurrent specification for the new algorithm in CSP, based on an existing specification of incremental minimization for the string case.
- Finally, Section 7 presents some concluding remarks and suggestions for future work.

2 Preliminaries

Since our discussion of minimization of tree automata frequently refers to that for the case of string automata, we recall some definitions related to *deterministic finite (string) automata*.

Definition 1. *A* DFA M *is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *such that* Q *is a finite set, the* state set*;* Σ *is an* alphabet *(a finite set of symbols);* $\delta \in Q \times \Sigma \nrightarrow Q$ *is the transition function;* $q_0 \in Q$ *is the* initial *or* start state*; and* $F \subseteq Q$ *is the set of* final *or* accepting states.

We extend transition function δ to its transitive closure δ^*, defined inductively by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$. For every state $q \in Q$ of a DFA, its *right language* (*left language*) is defined as $\overrightarrow{\mathcal{L}}= \{w \in \Sigma^* | \delta^*(q, w) \in F\}$ ($\overleftarrow{\mathcal{L}}= \{w \in \Sigma^* | \delta^*(q_0, w) = q\}$).

We call a DFA *complete* if and only if its transition function δ is a total function. A DFA with a partial transition function can be transformed into a complete one by adding a single *absorption state* or *sink state*, usually denoted $\perp$, and having undefined transitions lead to this state. A state $q \in Q$ is called *unreachable* if and only if its left language is empty. A DFA without unreachable states is called *reduced*. For simplicity, we assume DFAs to be both reduced and complete from here on.

Many of the notations and definitions we use are related to regular tree language theory. To a large extent they are straightforward generalizations of ones familiar from regular string language theory. Readers may want to consult e.g. [5,6,8,9] for more detail.

Let Σ be an alphabet, and $r \in \Sigma \to \mathbb{N}$. Pair (Σ, r) is a *ranked alphabet*, r is a *ranking function*, and for all $a \in \Sigma$, $r(a)$ is called the *rank* or *arity* of a. We use Σ_n for $0 \leq n$ to indicate the subset of Σ of symbols with arity n. In algorithms and predicates, we will use n to indicate the rank or arity of a symbol a.

Given a ranked alphabet (Σ, r), the set of *ordered, ranked trees* over this alphabet, set $Tr(\Sigma, r)$, is the smallest set satisfying $a \in Tr(\Sigma, r)$ for all $a \in \Sigma_0$ and $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ for all $t_1, \ldots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$. Such trees can trivially be presented as rooted, connected, directed, acyclic graphs in which each node has at most one incoming edge. Nodes labeled by symbols of rank 0 are called *leaf nodes* or *leaves*; the sequence of leafs of a tree is called its *frontier*.

In one common view on processing of a tree by tree automata (TAs), each tree node is annotated with a state. For each node labeled by a symbol a (of rank n), state q_0 and states $q_1, \cdots, q_n$ may be assigned to that node and its direct subnodes respectively if the tuple $(q_0, (q_1, \cdots q_n))$ is in the transition relation of symbol a. Note that this simplifies to $(q_0, ())$ for $n = 0$. A tree is *accepted* by a TA if and only if it can be consistently annotated such that the state assigned to the root is a so-called *root accepting* or final state.

By considering transitions of TAs to be directed Frontier-to-Root, we obtain the nondeterministic εNFRTAs; the deterministic DFRTAs are obtained by further restricting the automata to have no ε-transitions and by restricting the transition relations to be (partial) functions, i.e. for every state tuple and symbol yielding (at most) one state. This motivates the following definition:

Definition 2. *A* DFRTA *is a 5-tuple* $(Q, \Sigma, r, R, Q_{ra})$ *such that* Q *is a finite set, the* state set*;* (Σ, r) *is a ranked alphabet;* $R = \{R_a | a \in \Sigma\}$ *is the set of transition functions (where for all* $a \in \Sigma$ *with* $r(a) = n$ *we have* $R_a \in Q^n \nrightarrow Q$*); and* $Q_{ra} \subseteq Q$ *is the set of* root accepting *or* final states.

Compared to DFAs on strings, two main differences appear: DFRTAs have no start states, and each transition on a symbol (of rank or arity n) relates an n-tuple of states

to a state, instead of relating a single state to a state. Note that we will sometimes refer to Q_{ra} as F.

Just as the extension, δ^*, of a DFA's transition function δ (δ yields the state reached after processing a single symbol) yields the state reached after processing a string, for a DFRTA we can define a function RSt yielding the state reached after processing a tree, i.e. the state assigned to the root node of such a tree. It is defined inductively by $RSt(a) = R_a()$ for $a \in \Sigma_0$ and $RSt(a(t_1, \ldots, t_n)) = R_a(RSt(t_1), \ldots, RSt(t_n))$ for $t_1, \ldots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$.

Using this definition, we can define the language accepted at state q in a DFRTA M as $\mathcal{L}^{\downarrow}_M(q) = \{t \in Tr(\Sigma, r) | RSt(t) = q\}$. The language accepted by the DFRTA then is simply the language accepted at either of its final or root accepting states: $\mathcal{L}_M = \bigcup_{q \in Q_{ra}} \mathcal{L}^{\downarrow}_M(q)$. If M is clear from the context, we simply write $\mathcal{L}$ instead of $\mathcal{L}_M$. Note that $\mathcal{L}^{\downarrow}_M(q)$ is analogous to the left language of a DFA state [7]; for obvious reasons, we will therefore call it the *down language* of state q. Likewise, we define the *up language* of a state, a notion similar to the right language of a DFA state [7]: $\mathcal{L}^{\uparrow}_M(q) = \{t \in Tr(\Sigma', r') | RSt(t \cdot_{\#} s) \in F \text{ for all } s \in \mathcal{L}^{\downarrow}_M(q)\}$ where t has a single leaf labeled # (and Σ' and r' are obtained by extending Σ and r with this symbol of arity 0) and $t \cdot_{\#} s$ denotes the tree obtained from t by substituting tree s for this leaf.

We call a DFRTA *complete*, similar to the notion of completeness for DFAs, if and only if the R_a are total functions; a DFRTA that is not complete can always be made complete by adding a sink state and transitions to it, as is the case for DFAs. A DFRTA state q is *unreachable* if and only if it can never be assigned to the root of any tree by a computation of the DFRTA—i.e. if and only if its down language $\mathcal{L}^{\downarrow}_M$ is empty. Like DFAs, DFRTAs are *reduced* if and only if they contain no unreachable states. From here on, we assume DFRTAs to be both reduced and complete.

Finally, the size of a DFA or DFRTA is defined as $|Q|$, i.e. the size of its state set.

3 Equivalence and minimality

We use *Equiv* as a predicate on two DFA states, defined for all $p, q \in Q$ by

$$Equiv(p, q) \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)).$$

and as a predicate on two DFRTA states, defined for all $p, q \in Q$ by

$$Equiv(p, q) \equiv (\mathcal{L}^{\uparrow}(p) = \mathcal{L}^{\uparrow}(q)).$$

Thus DFA (DFRTA) states are equivalent if and only if their right languages (up languages) are the same. Using an inductive definition of $\overrightarrow{\mathcal{L}}$, $Equiv(p, q)$ for DFA states can easily be defined recursively [13] as

$$(p \in F \equiv q \in F) \wedge \left\langle \forall\, a : a \in \Sigma : Equiv(\delta(p, a), \delta(q, a)) \right\rangle .$$

Similarly, $Equiv(p, q)$ can be defined recursively for DFRTA states. Before doing so, we introduce two abbreviations:

- $\overrightarrow{\rho}_{i:s}$ is used to abbreviate $(\rho_1, \ldots, \rho_{i-1}, s, \rho_{i+1}, \ldots, \rho_n)$ (given $\overrightarrow{\rho} = (\rho_1, \ldots, \rho_n)$);
- Predicate $P(a, i, \overrightarrow{\rho})$ is defined as $a \in \Sigma \wedge 1 \leq i \leq n \wedge \overrightarrow{\rho} \in Q^n$.

Using these abbreviations $Equiv(p, q)$ can be defined recursively for DFRTA states as

$$(p \in Q_{ra} \equiv q \in Q_{ra}) \wedge \left\langle \forall\, a, i, \vec{\rho} \colon P(a, i, \vec{\rho}) : Equiv(R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q})) \right\rangle$$

(proof omitted and similar to the string case, albeit slightly more complicated due to the generalisation from string right languages to tree up languages). In other words, two DFRTA states are equal if and only if they are both final or non-final, and for each alphabet symbol and each two state tuples that are identical except for the appearance of p and q in corresponding positions, the transitions from these two tuples on the symbol lead to equivalent states.

The usual definition of minimality of a finite automaton (whether on strings or on trees), is that no language equivalent automaton with fewer states exists. Using the definition of up language for DFRTA states (respectively right language for DFA states), minimality can also be written as a predicate

$$\left\langle \forall\, p, q \in Q : p \neq q : \neg Equiv(p, q) \right\rangle .$$

For any two states p, q (such that $p \neq q$), if $Equiv(p, q)$ holds, they can be merged, i.e. one of them can be eliminated in favor of the other (while redirecting in-transitions to the eliminated state to the equivalent remaining one). Eventually, the resulting automaton will be the minimal one recognizing the same language as the original one. (Note that this minimal DFA or DFRTA is unique up to isomorphism). We do not address this reduction step in this paper, but focus on the computation of *Equiv* in two essentially different ways.

4 The classical minimization approach

In the classical approach, the computation of *Equiv* starts out from two initial partitions, corresponding to F and $Q \backslash F$. This is refined iteratively until the greatest fixed-point is reached. The resulting partitioning corresponds to the state set of the minimal automaton equivalent to the original one. Put differently, the algorithms compute the greatest fixed point from the top i.e. from the unsafe side [13].

Classically, minimization algorithms may in fact be based on computing the distinguishability relation between states instead of the equivalence relation, or on computing the partition induced on states by the equivalence relation, or some combination of the three. One variant of the classical minimization approach *for the case of* DFA*s* is presented in [12, Algorithm 7.18]. It uses layerwise computation of *Equiv* (called E there) and of its negation $\neg Equiv$ (called D there, and not included in our presentation). We slightly adapt that algorithm to our notation here:

Algorithm 3 (Layerwise computation of *Equiv* for DFAs)

$H := (F \times F) \cup ((Q \backslash F) \times (Q \backslash F))$;
$H_{old} := Q \times Q$;
{ invariant: $H \supseteq Equiv$ }
do $H \neq H_{old} \rightarrow$
  { $H \neq H_{old}$ }
  $H_{old} := H$;
  for $(p, q) : (p, q) \in H_{old} \rightarrow$
    as $\langle \exists\, a : a \in \Sigma : (\delta(p, a), \delta(q, a)) \notin H_{old} \rangle \rightarrow H := H \backslash (p, q)$ **sa**
  rof
od{ $H = Equiv$ }

As pointed out by Watson, without the computation of $\neg Equiv$ (i.e. D), this is essentially Wood's algorithm for computing minimal DFAs [14, p. 132], with Wood stating it is based on Moore's 1950s work [10].

A version of this algorithm *for the case of* DFRTA*s* is presented below. It essentially corresponds to the approach in [6, Section 1.5].[1] The resulting equivalence relation *Equiv* (called P there) is then used to determine the induced equivalence classes and construct the corresponding minimal DFRTA.

Algorithm 4 (Layerwise computation of *Equiv* for DFRTAs)

$H := (F \times F) \cup ((Q \backslash F) \times (Q \backslash F))$;
$H_{old} := Q \times Q$;
{ invariant: $H \supseteq Equiv$ }
do $H \neq H_{old} \rightarrow$
  { $H \neq H_{old}$ }
  $H_{old} := H$;
  for $(p, q) : (p, q) \in H_{old} \rightarrow$
    as $\left\langle \exists\, a, i, \vec{\rho} \colon P(a, i, \vec{\rho}) : (R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q})) \notin H_{old} \right\rangle \rightarrow H := H \backslash (p, q)$ **sa**
  rof
od{ $H = Equiv$ }

Even though this algorithm may look rather complicated compared to the one for DFAs, there is only one essential difference: instead of considering for each symbol a and state p and q where their out-transition on this symbol leads, one has to consider this for states p and q within contexts: their occurrence at the same position in two otherwise equal state tuples (cf. the definition of *Equiv* for DFRTA states as given at the end of Section 3).

The classical minimization approach for DFRTAs has been known for decades, with its first description appearing in Brainerd's 1967 PhD thesis [1]. That description is not as explicitly algorithmic as the one given here or in [6, Section 1.5], and in fact, an algorithmic presentation worked out in detail to an implementation level—albeit for the case of DFRTAs on *unranked* trees—did not appear until 2007 [3].

[1] Note however, that the quantification used in [6, Section 1.5] is somewhat imprecise, as it leaves i unbounded.

5 An incremental minimization algorithm

Watson in [12] and most recently Watson & Daciuk in [13] presented an incremental approach to DFA minimization. Their approach results in an algorithm that starts out with a singleton partition for each of the states of the initial DFA and refines this partition by iteratively merging partitions that are shown to be equivalent. The greatest fixed-point reached corresponds to the state set of the minimal automaton equivalent to the original one. Such an algorithm thus computes the fixed point from below i.e. from the safe side. Clearly, intermediate results from such an algorithm can already be used to reduce the original DFA. We provide a first version of this incremental approach for DFRTAs.

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences *Equiv* into a functional program. For cyclic automata, a direct translation from definition to functional program might lead to non-termination. Thus, in addition to two states, the functional program for compute equivalence also takes a third parameter. An invocation $equiv(p, q, \emptyset)$ returns, via the local variable eq, the truth value of $Equiv(p, q)$. The third parameter, S, is used during recursion to capture pairs of states that are assumed to be equivalent until shown otherwise.

The recursion depth can be bounded by the larger of $|Q| - 2$ and 0 without affecting the result [12, Section 7.3.3], and we add a parameter k to function *equiv* to do so. For efficiency reasons, parameter S is made a global variable. We assume that it is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q| - 2)$ **max** $0)$ returns $Equiv(p, q)$; after such an invocation returns, $S = \emptyset$.

Algorithm 5 (Pointwise computation of $Equiv(p, q)$ for DFAs)

func $equiv(p, q, k) =$
⟦ **if** $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
⟦ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
⟦ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
$eq := (p \in F \equiv q \in F)$;
$S := S \cup \{\{p, q\}\}$;
for $a : a \in \Sigma \rightarrow$
$eq := eq \wedge equiv(\delta(p, a), \delta(q, a), k - 1)$
rof;
$S := S \backslash \{\{p, q\}\}$
fi;
return eq
⟧{ $equiv(p, q, k) \equiv Equiv(p, q)$ }

Function *equiv* can be used to compute relation *Equiv*. To do so, we maintain set G (H) consisting of pairs of states known to be distinguishable i.e. belonging to $\neg Equiv$ (equivalent i.e. belonging to *Equiv*). To initialize both sets, we note that final states are never equivalent to non-final ones, and that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, we ensure that H is transitive at each step of the algorithm. Finally, we have a global variable S as used by function *equiv*:

Algorithm 6 (Incremental computation of *Equiv*)

$S, G, H := \emptyset, ((Q \backslash F) \times F) \cup (F \times (Q \backslash F)), \{(q, q) | q \in Q\};$
{ invariant: $G \subseteq \neg Equiv \wedge H \subseteq Equiv$ }
do $(G \cup H) \neq Q \times Q \rightarrow$
  let $p, q : (p, q) \in ((Q \times Q) \backslash (G \cup H))$;
  if $equiv(p, q, (|Q| - 2)$ **max** $0) \rightarrow$
    $H := H \cup \{(p, q), (q, p)\}$;
    $H := H^+$
  [] $\neg equiv(p, q, (|Q| - 2)$ **max** $0) \rightarrow$
    $G := G \cup \{(p, q), (q, p)\}$;
  fi
od{ $H = Equiv$ }

The repetition in this algorithm can be interrupted and the partially computed H can be safely used to merge states, leading to a not necessarily minimal but potentially smaller automaton than the original one.

The algorithm is not DFA-specific and as a result can be applied for the DFRTA-case, provided function *equiv* is suitably chosen. Looking at function *equiv* for the DFA case, we see that the update to *eq* in the loop is performed for every out-transition of p and q. For the DFRTA case, the equivalent is to perform the update for every out-transition *involving* p and q, with such out-transitions involving tuples of states that are identical except for an appearance of p and q respectively at the same position:

Algorithm 7 (Pointwise computation of $Equiv(p, q)$ for DFRTAs)

func $equiv(p, q, k) =$
|[**if** $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  [] $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  [] $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F)$;
    $S := S \cup \{\{p, q\}\}$;
    for $a, i, \vec{\rho}$: $P(a, i, \vec{\rho}) \rightarrow$
      $eq := eq \wedge equiv(R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q}), k - 1)$
    rof;
    $S := S \backslash \{\{p, q\}\}$
  fi;
  return eq
]|{ $equiv(p, q, k) \equiv Equiv(p, q)$ }

Watson and Daciuk in [13] considered different ways to improve both the theoretical and the practical running time of the algorithm for the DFA case. Furthermore, they showed the resulting efficient implementation to be competitive to implementations of classical minimization algorithms, even though the basic incremental algorithm is known to have a worse theoretical running time complexity. We expect similar results to hold for the DFRTA case and plan to consider these as future work.

6 A CSP specification for incremental DFRTA minimization

In [11, Section 5], Strauss et al. presented a concurrent version of the incremental minimization algorithm for DFAs in the form of a CSP specification. The crucial part of that specification w.r.t. the difference between DFAs and DFRTAs is presented below.[2] It corresponds to the **for**-loop in function *equiv* of Algorithm 5.

$$
\begin{aligned}
FanOut_{pq}(S,k) = {} & |||_{a\in\Sigma} \\
& (\text{if } (\{\delta(p,a),\delta(q,a)\} \notin S) \text{ then} \\
& \quad (\ Equiv_{\delta(p,a),\delta(q,a)}(S \cup \{(p,q)\}, k-1)\ \triangle \qquad (1)\\
& \quad (to_{\delta(p,a),\delta(q,a)}?eq_a \rightarrow (EqSet := EqSet \cup \{eq_a\})\ ) \\
& \text{else } (EqSet := EqSet \cup \{true\})\ ) \qquad (2)
\end{aligned}
$$

We refer to [11, Section 5] for details on this and other parts of the specification. Here, we focus on adapting this particular part to the case of DFRTAs. All the other parts of the specification stay the same, just as all the other parts of the sequential algorithm stay the same when generalizing from DFAs to DFRTAs (compare Algorithm 5 to Algorithm 7).

To generalize the specification of $FanOut_{pq}(S,k)$, we merely need to generalize the range of the interleaving operator $|||$ from $a \in \Sigma$ to $P(a,i,\vec{\rho})$ and replace the $\delta(p,a)$ and $\delta(q,a)$ by $R_a(\vec{\rho}_{i:p})$ and $R_a(\vec{\rho}_{i:q})$.

The generalization of the CSP specification from DFAs to DFRTAs is thus rather elegant. As hinted at in [11], a significant advantage of a CSP specification such as the foregoing, is maximally to expose opportunities for parallelization. Expressing the DFRTA minimization algorithm in the suggested CSP format indicates that these opportunities will increase drastically if there are a large number of state tuples (which in turn depends on the ranks of the symbols and the number of states). How one exploits these opportunities will clearly depend on the available hardware configuration. The CSP specification is provided in anticipation of a continuation in the current surge in the chip industry towards increasingly large multi-core processors. Thus, while in some senses the CSP specification is a theoretical result, we believe that it is sufficiently generic to serve as a useful reference point in experimenting with parallel implementations of the DFRTA minimization algorithm.

7 Conclusion

This paper has high-lighted once again that many results from the field of regular string languages generalize to that of regular tree languages. It showed, by way of three minimization algorithms, how this generalization becomes quite transparent and elegant if suitable notation is used.

The first algorithm that was generalized to the DFRTA case was already known, but has been presented here in a style which highlights how the generalization occurs.

The second algorithm generalized to the DFRTA case gives a completely new result, being namely a generalization to ranked trees of the string algorithm presented by Watson and Daciuk, incrementally minimizing a DFRTA. As a result, intermediate results of the algorithm can be used to reduce the initial automaton's size. This makes

[2] The specification has been slightly adapted to the notation used in the current paper.

the algorithm useful in situations where running time is restricted (for example, in real-time applications). The new incremental minimization algorithm for DFRTAs can be further improved, similar to the improvements made for the DFA case in [13, Section 6]. We expect such improvements to lead to better performance in practice, similar to the DFA case. To verify this and to be able to compare the (improved) new algorithm and the one using a classical approach to minimization, both need to be implemented and benchmarked.

In the third instance, we also briefly described how an existing concurrent specification of the incremental DFA minimization algorithm in CSP gives rise to one for the DFRTA case. Once again, the generalization was facilitated by relying on suitably defined notation. While implementations of the concurrent specification could be investigated to see whether the parallelization is efficient in practice on currently available hardware, we consider that its principal value lies in serving as a reference point in deriving parallel implementations on the anticipated massively parallel machines of the future.

References

1. W. S. Brainerd: *Tree Generating Systems and Tree Automata*, PhD thesis, Purdue University, June 1967.
2. W. S. Brainerd: *The minimalization of tree automata*. Information and Control, 13(5) November 1968, pp. 484–491.
3. R. C. Carrasco, J. Daciuk, and M. L. Forcada: *An implementation of deterministic tree automata minimization*, in CIAA, J. Holub and J. Zdárek, eds., vol. 4783 of Lecture Notes in Computer Science, Springer, 2007, pp. 122–129.
4. R. C. Carrasco, J. Daciuk, and M. L. Forcada: *Incremental construction of minimal tree automata*. Algorithmica, 2008.
5. L. G. W. A. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, April 2008, `http://alexandria.tue.nl/extra2/200810270.pdf`.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata: Techniques and applications*, 2007, `http://www.grappa.univ-lille3.fr/tata/`.
7. J. Daciuk and R. C. Carrasco: *Perfect hashing with pseudo-minimal bottom-up deterministic tree automata*, in Intelligent Information Systems XVI, Proceedings of the International IIS'08 Conference held in Zakopane, Poland, June 16-18, 2008, M. A. Klopotek, A. Przepiorkowski, S. T. Wierzchon, and K. Trojanowski, eds., Academic Publishing House Exit, Warszawa, 2008, pp. 229–238.
8. J. Engelfriet: *Tree Automata and Tree Grammars*, Lecture Notes DAIMI FN-10, Aarhus University, April 1975.
9. F. Gécseg and M. Steinby: *Tree Automata*, Akadémiai Kiadó, Budapest, 1984.
10. E. F. Moore: *Gedanken experiments on sequential machines*, in Automata Studies, C. E. Shan and J. McCarthy, eds., Princeton University Press, Princeton, NJ, 1956.
11. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of an incremental DFA minimisation algorithm*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 218–226.
12. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Dept. of Mathematics and Computing Science, Technische Universiteit Eindhoven, September 1995, `http://www.fastar.org/publications/PhD_Watson.pdf`.
13. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm*. Natural Language Engineering, 9(1) 2003, pp. 49–64.
14. D. Wood: *Theory of Computation*, Harper & Row, New York, 1987.