Information and Computation 207 (2009) 1181–1208

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Efficient inclusion checking for deterministic tree automata and XML schemas

Jérôme Champavère [a,b,*,1], Rémi Gilleron [a,b,1], Aurélien Lemay [a,b,1], Joachim Niehren [a,1]

[a] *Inria Lille – Nord Europe, Parc scientifique de la Haute Borne, 40 avenue Halley, Bât. A, Park Plaza, 59650 Villeneuve d'Ascq, France*

[b] *Laboratoire d'Informatique Fondamentale de Lille, Université de Lille, France*

ARTICLE INFO

Article history:
Received 11 June 2008
Revised 3 March 2009
Available online 24 March 2009

Keywords:
Tree automata
Language inclusion
Algorithmic complexity
XML schemas

ABSTRACT

We present algorithms for testing language inclusion $L(A) \subseteq L(B)$ between tree automata in time $O(|A| \cdot |B|)$ where B is deterministic (bottom-up or top-down). We extend our algorithms for testing inclusion of automata for unranked trees A in deterministic DTDs or deterministic EDTDs with restrained competition D in time $O(|A| \cdot |\Sigma| \cdot |D|)$. Previous algorithms were less efficient or less general.

1. Introduction

Language inclusion for tree automata is a basic decision problem that is closely related to universality and equivalence [1–3]. Tree automata algorithms are generally relevant for XML document processing [4–7]. Regarding inclusion, typical applications are inverse type checking for tree transducers [8] and schema-guided query induction [9]. The latter was the motivation for the present study. There, candidate queries produced by the learning process are to be checked for consistency with deterministic DTDs, such as for HTML.

We investigate language inclusion $L(A) \subseteq L(B)$ for tree automata A and B under the assumption that B is bottom-up deterministic or top-down deterministic, not necessarily A. Without this assumption, the problem becomes DEXPTIME-complete [3]. Deterministic language inclusion still subsumes universality of deterministic tree automata $L(B) = T_\Sigma$ up to a linear time reduction, as well as equivalence of languages of two deterministic automata $L(A) = L(B)$. Conversely, one can reduce inclusion to equivalence in PTIME, since $L(A) \subseteq L(B)$ if and only if $L(A) \cap L(B) = L(A)$. However, this leads to a reduction in quadratic time $O(|A| \cdot |B|)$, so we cannot rely on equivalence testing (as by comparing cardinalities [2] or unique minimal deterministic automata) for efficient inclusion testing.

The well-known naive test for inclusion in bottom-up deterministic tree automata for ranked trees goes through complementation. It first computes an automaton B^c that recognizes the complement of the language of B, and then checks whether the intersection automaton for B^c and A has a non-empty language. The problematic step is the completion of B before complementing its final states, since completion might require adding rules for all possible left-hand sides. The overall running time may thus become $O(|A| \cdot |\Sigma| \cdot |B|^n)$, which is exponential in the maximal rank n of function symbols in the signature Σ. This time complexity can be reduced by turning the maximal arity of function symbols in ranked trees to 2.

* Corresponding author.
E-mail address: champavere@grappa.univ-lille3.fr (J. Champavère).

[1] Mostrare project http://mostrare.futurs.inria.fr/, Inria Lille–Nord Europe.

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2009.03.003

It is folklore that one can transform ranked trees into binary trees, and automata correspondingly. The problem here is to preserve bottom-up determinism, while the size of automata must remain linear. We can solve this problem by using Curried encodings of unranked tree into binary trees, as proposed for stepwise tree automata [10,1]. Thereby we obtain an inclusion test for the ranked case in time $O(|A| \cdot |\Sigma| \cdot |B|^2)$. This is still too much in practice with XML Schemas, where A and B may be of size 500 and Σ of size 100; these orders of magnitude can be observed, e.g., in the DTDs of the corpus studied by Bex et al. [11].

Our first contribution is a more efficient algorithm that test inclusion in bottom-up deterministic tree automata in time $O(|A| \cdot |B|)$. This bound is independent of the size of the signature Σ, even if it is not fixed. We establish our algorithm for stepwise tree automata over binary trees in the first step, and then lift it *via* currying to standard tree automata for ranked trees over arbitrary signatures and to stepwise tree automata over unranked trees.

As a second contribution, we show how to test language inclusion of stepwise tree automata A for unranked trees in deterministic DTDs D in time $O(|A| \cdot |\Sigma| \cdot |D|)$. Determinism for DTDs is required by the XML standards. Our algorithm first computes Glushkov automata for all regular expressions in D in time $O(|\Sigma| \cdot |D|)$. This is possible since we assume D to be deterministic DTDs [12]. The second step is more tedious. We would like to transform the whole collection of Glushkov automata into a single bottom-up deterministic stepwise tree automaton of the same size. Unfortunately, this seems difficult to achieve, since the usual construction of Martens and Niehren [13] eliminates ϵ-rules on the fly, which may lead to a quadratic blowup of the number of rules (not the number of states).

We solve this problem by introducing bottom-up deterministic *factorized tree automata*. These are tree automata with ϵ-rules, which represent deterministic stepwise tree automata more compactly, and in particular the collection of Glushkov automata of a DTD of linear size. Factorized tree automata have two sorts of states, which play the roles of hedge and tree states in alternative automata notions for unranked trees [14,4]. The difficulty is to define the appropriate notion of determinism for factorized tree automata, and to adapt the inclusion test to the case where B is a deterministic factorized tree automaton.

Our results can be applied if A is a hedge automaton [1,15,16], with finite word automata for horizontal languages, since such hedge automata can be translated in linear time to stepwise tree automata. Note, however, that the notion of (bottom-up) determinism for hedge automata is unsatisfactory [13] so that we cannot choose B to be a deterministic hedge automaton even if the horizontal language is defined by a deterministic finite word automaton.

The situation becomes slightly different if A is a tree automaton recognizing firstchild-nextsibling encodings of unranked trees, and D a DTD. The problem is that the conversion of A into a stepwise tree automaton may lead to a quadratic size increase. In this case, however, we can encode DTDs into top-down deterministic tree automata that recognize firstchild-nextsibling encodings of unranked trees, and reduce the inclusion problem to the case of inclusion in deterministic finite word automata. This yields a worst case running time of $O(|A| \cdot |\Sigma| \cdot |D|)$, too. As we show, the same algorithm applies if D is a deterministic extended DTD (EDTD) with restrained competition [17]. These were introduced in order to reason about schema definitions in the W3C standard XML Schema [18,19], and relaxations thereof.

Our algorithm generalizes the inclusion test of Martens et al. [18] (see Section 10 of the reference), where A and D are both limited to deterministic EDTDs with restrained competition. The presentation of our algorithm differs in that we rely on inclusion in top-down deterministic tree automata *via* firstchild-nextsibling encoding as an intermediate step, while they reduce the problem to inclusion in deterministic finite word automata directly (*via* the main theorem of this article). Furthermore, we provide a precise complexity analysis for the first time (which is not fully obvious).

Related Work. Our new algorithm for testing inclusion in bottom-up deterministic factorized tree automata B is relevant for schemas defined in Relax NG [20]. This holds for those definitions that can be made bottom-up deterministic without combinatorial explosion. Furthermore, we can permit arbitrary Relax NG schemas as automata A on the left, where no determinism is required.

The folklore algorithms for testing inclusion in top-down deterministic automata (by reduction to finite automata for path languages) lead us to a generalization of the inclusion test for deterministic restrained competition EDTDs [18]. This is useful for testing inclusion of Relax NG in XML Schema for instance.

Compared to the conference version of the present article at LATA'08 [21], we have added new results on early failure detection, incrementality, experiments, and complete proofs. Furthermore, we added the alternative algorithm for inclusion in top-down deterministic automata and in restrained competition EDTDs. We simplified the presentation of our algorithms in many places. Meanwhile, the inclusion test presented here has been integrated into a system for schema-guided query induction [9], where it proves its efficiency in practice.

The complexity of inclusion for various fragments of DTDs and EDTDs was first studied by Martens et al. [22]. They assume the same types of language definitions on both sides. When applied to deterministic DTDs, the same complexity results seem obtainable when refining the efficiency analysis provided there. In any case, our algorithm permits richer left-hand sides (as needed in schema-guided query induction) without increasing in complexity.

Heuristic algorithms for inclusion between non-deterministic automata and applications that avoid the high worst-case complexity were proposed by Tozawa and Hagiya [23]. Even though motivated by XML Schema, for which better algorithms are available meanwhile (due to top-down determinism), they are relevant for Relax NG where no notion of determinism is imposed *a priori*.

Outline. In Section 2, we reduce inclusion for ranked tree automata to the binary case. An efficient incremental algorithm for binary tree automata is given in Section 3. In Section 4, we introduce deterministic factorized tree automata and lift the inclusion test. In Section 5, we apply it to test inclusion of automata in deterministic DTDs. Section 6 presents experimental results. Section 7 studies inclusion in top-down deterministic tree automata and restrained competition extended DTDs. Appendix 8 details the implementation.

2. Standard tree automata for ranked trees

We reduce the inclusion problem of tree automata for ranked trees [1] to the case of binary trees with a single binary function symbol.

A ranked signature Σ is a finite set of function symbols $f \in \Sigma$, each of which has an arity $ar(f) \geq 0$. A constant $a \in \Sigma$ is a function symbol of arity 0. A tree $t \in T_\Sigma$ is either a constant $a \in \Sigma$ or a tuple $f(t_1, \ldots, t_n)$ consisting of a function symbol f with $ar(f) = n$ and trees $t_1, \ldots, t_n \in T_\Sigma$.

A *tree automaton* A over Σ with ϵ-rules consists of a finite set $sta(A)$ of states, a subset $fin(A) \subseteq sta(A)$ of final states, and a set $rul(A) \subseteq sta(A)^2 \uplus (\cup_{n\geq 0}\{f \in \Sigma \mid ar(f) = n\} \times sta(A)^{n+1})$. We denote such rules as $p' \xrightarrow{\epsilon} p$ or $f(p_1, \ldots, p_n) \to p$, where $f \in \Sigma$ has arity n and $p_1, \ldots, p_n, p, p' \in sta(A)$. Furthermore, we write $p' \xrightarrow{\epsilon}_A p$ iff $p' \xrightarrow{\epsilon} p \in rul(A)$, $\xrightarrow{\epsilon}{}^*_A$ for the reflexive transitive closure of $\xrightarrow{\epsilon}_A$, and $\xrightarrow{\epsilon}{}^{\leq 1}_A$ for the union of $\xrightarrow{\epsilon}_A$ and the identity relation on $sta(A)$.

The *size* of an ϵ-rule $p \xrightarrow{\epsilon} p'$ is 2 and that of a rule $f(p_1, \ldots, p_n) \to p$ is $n + 2$. The *size* $|A|$ of A is the cardinality of $sta(A)$, denoted $|sta(A)|$, plus the sum of the sizes of the rules of A, which we denote $|rul(A)|$. The cardinality $|\Sigma|$ of the signature Σ is ignored, since it is irrelevant for algorithms that care only about used symbols. Every tree automaton A defines an evaluator $eval_A : T_{\Sigma\cup sta(A)} \to 2^{sta(A)}$ such that:

$$eval_A(f(t_1, \ldots, t_n)) = \{p \mid p_1 \in eval_A(t_1), \ldots, p_n \in eval_A(t_n), f(p_1, \ldots, p_n) \to p' \in rul(A),\ p' \xrightarrow{\epsilon}{}^*_A p\}$$

and $eval_A(p) = \{p\}$. A tree $t \in T_\Sigma$ is *accepted* by A if $fin(A) \cap eval_A(t) \neq \emptyset$. The *language* $L(A)$ is the set of trees accepted by A.

A tree automaton is (bottom-up) *deterministic* if it has no ϵ-rules, and if no two rules have the same left-hand side. It is complete if there are rules for all potential left-hand sides. It is well-known that deterministic complete tree automata can be complemented in linear time, by switching the final states.

Deterministic inclusion. We will study the deterministic inclusion problem for tree automata. Its input consists of a ranked signature Σ, a possibly non-deterministic tree automaton A with ϵ-rules, and a deterministic tree automaton B, both with signature Σ, and its output is the truth value of $L(A) \subseteq L(B)$.

We can deal with this problem by restriction to stepwise signatures $\Sigma_@$, which consist of a single binary function symbol @ and a finite set of constants $a \in \Sigma$. A stepwise tree automaton over binary trees is a tree automaton over a stepwise signature [10]. We use the infix notation in automata rules and thus write $q_1@q_2 \to q$ instead of $@(q_1, q_2) \to q$.

In Section 5, we will see how to interpret stepwise tree automata over unranked trees *via* binary encoding. Here, we use the same binary encoding for interpretation over ranked trees with arbitrary signatures.

Proposition 1. *The deterministic inclusion problem for standard tree automata over ranked trees can be reduced in linear time to the deterministic inclusion problem for stepwise tree automata over binary trees.*

We first encode ranked trees into binary trees *via* currying. Given a ranked signature Σ we define the corresponding signature $\Sigma_@ = \{@\} \uplus \Sigma$ whereby all symbols of Σ become constants. We use infix notation for the binary symbol @ and write $t_1@t_2$ instead of $@(t_1, t_2)$. Furthermore, we assume that omitted parentheses have priority to the left, i.e., we write $t_1@t_2@t_3$ instead of $(t_1@t_2)@t_3$. Currying is defined by a function $curry : T_\Sigma \to T_{\Sigma_@}$ which for all trees $t_1, \ldots, t_n \in T_\Sigma$ and $f \in \Sigma$ satisfies:

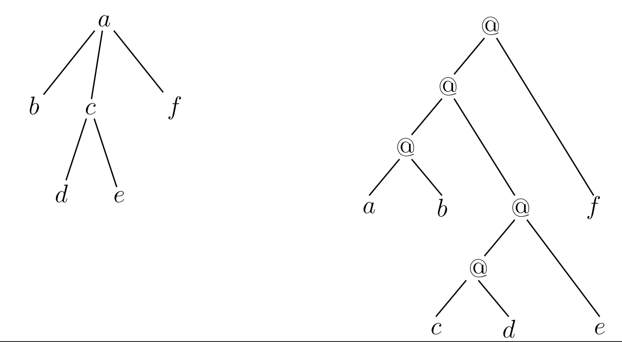


Fig. 1. Currying the ranked tree $a(b, c(d, e), f)$ into the binary tree $a@b@(c@d@e)@f$.

$$\frac{f(q_1,\ldots,q_n)\to q\in \mathrm{rul}(A)\qquad 1\le i<n}{\begin{array}{c}f\to f\in \mathrm{rul}(\mathrm{step}(A))\\ f q_1\ldots q_{i-1}@q_i\to f q_1\ldots q_i\in \mathrm{rul}(\mathrm{step}(A))\\ f q_1\ldots q_{n-1}@q_n\to q\in \mathrm{rul}(\mathrm{step}(A))\end{array}}\qquad \frac{a\to q\in \mathrm{rul}(A)}{a\to q\in \mathrm{rul}(\mathrm{step}(A))}$$

Fig. 2. Transforming ranked tree automata into stepwise tree automata.

$$curry(f(t_1,\ldots,t_n)) = f@curry(t_1)@\ldots@curry(t_n)$$

For instance, $a(b,c(d,e),f)$ is mapped to $a@b@(c@d@e)@f$, which is the infix notation for the tree $@(@(@(a,b),@(@(c,d),e)),f)$, as shown in Fig. 1.

Now we encode tree automata A over Σ into stepwise tree automata $step(A)$ over $\Sigma_@$, such that the language is preserved up to currying, i.e., such that $L(step(A)) = curry(L(A))$. The states of $step(A)$ are the prefixes of left-hand sides of rules in A, i.e., words in $\Sigma(sta(A))^*$:

$$sta(step(A)) = \{fq_1\ldots q_i \mid f(q_1,\ldots,q_n)\to q\in rul(A),\ 0\le i\le n\}\uplus sta(A)$$

The rules of $step(A)$ are given in Fig. 2. They extend prefixes step by step by states q_i according to the rules of A. Since constants cannot be extended, we need to distinguish two cases.

Lemma 2. *The encoding of tree automata A over Σ into stepwise tree automata $step(A)$ over $\Sigma_@$ preserves determinism, the tree language modulo currying, and the automata size up to a constant factor of* 3.

As a consequence, $L(A)\subseteq L(B)$ is equivalent to $L(step(A))\subseteq L(step(B))$, and can be tested in this way modulo a linear time transformation. Most importantly, the determinism of B carries over to $step(B)$.

3. Stepwise tree automata for binary trees

We present our new algorithm for testing deterministic inclusion in the case of stepwise tree automata over binary trees. We start with a characterization of deterministic inclusion, express it by a Datalog program [24,25], and then turn it into an efficient algorithm, which is non-trivial.

3.1. Ground Datalog

For the sake of self-containedness, we recall folkore results on ground Datalog (see, e.g., Gottlob et al. [26]). A ground Datalog program is set of Horn clauses, without function symbols, variables, and negation. More formally, it is build from a ranked signature Γ with constants $c\in\Gamma$ and predicates $\mathrm{p}\in\Gamma$, each of which has an arity $ar(\mathrm{p})\ge 0$. A literal is a term of the form $\mathrm{p}(c_1,\ldots,c_{ar(\mathrm{p})})$. We write $lit(\Gamma)$ for the set of all literals over Γ. A (Horn) clause, written as "L :– $L_1,\ldots,L_k$.", is a pair in $lit(\Gamma)\times lit(\Gamma)^k$, where $k\ge 0$. As usual, we write "L." instead of "L :– .", where $k=0$. A ground Datalog program P over Γ is a finite set of Horn clauses over Γ. The size $|P|$ of a Datalog program P is the overall number of occurrences of symbols in its clauses.

Every ground Datalog program P over Γ has a unique least fixed point $lfp(P)$ (since there is no negation). This is the least set of literals over Γ that satisfies for all L :– $L_1,\ldots,L_k$. in P, that $L_1\ldots,L_k\in lfp(P)$ implies $L\in lfp(P)$. Least fixed points are always finite sets (in the absence of function symbols).

The next theorem states that least fixed points can be computed efficiently (since there are no variables).

Theorem 3 (Efficiency of ground Datalog). *For every signature Γ and ground Datalog program P over Γ, the least fixed point $lfp(P)$ can be computed in linear time $O(|P|)$.*

Note that the upper bound $O(|P|)$ may depend on the arities of predicates in P, but not on the arities of the other predicates of Γ.

Proof. A program P defines a hypergraph, whose edges are the tuples $(L,L_1,\ldots,L_k)$ with L :– $L_1,\ldots,L_k$. in P. The least fixed point $lfp(P)$ is the set of literals accessible in this hypergraph. It is well-known that accessible components of graphs can be computed in linear time. The same holds for hypergraphs, under the often implicit condition that $L=L'$ can be tested in time $O(1)$. This condition is clearly valid if all predicates in P have arity 0. For this case, the theorem is folklore (see, e.g., Minoux [27]).

For an arbitrary signature Γ, we consider $lit(\Gamma)$ as the ranked signature without constants, in which all literals become predicates of arity 0. Literals L over Γ corresponds one-to-one to literals $L()$ over $lit(\Gamma)$. Thus, every ground Datalog program over Γ can be transformed into a ground Datalog program over $lit(\Gamma)$ in time $O(|P|)$. In order to test $L=L'$ in time $O(1)$ as required above, we have to replace all literals in P by numbers, such that different occurrences of the same literal are mapped to the same number. This can be done in time $O(|P|)$ by using a prefix tree, that memorizes all the numbers assigned to literals seen so far. For instance, the prefix-tree $lit(\mathsf{p}_1(c_1(c_2(1), c_3(2)), \mathsf{p}_0(c_1(3))))$ memorizes the assignments of $\mathsf{p}_1(c_1, c_2)$ to 1, of $\mathsf{p}_1(c_1, c_3)$ to 2, and of $\mathsf{p}_0(c_1)$ to 3. $\square$

We can refine least fixed points from sets to multisets, by counting for every literal the multiplicity with which it can be added to the least fixed point, where $\#S$ denotes the cardinality of the set S:

$$\begin{aligned} &lfp^{\#}(P) : lit(\Gamma) \to \mathbb{N} \cup \{0\} \\ &lfp^{\#}(P)(L) = \#\{R \in lfp(P)^k \mid L \coloneq R. \text{ in } P,\ k \geq 0\} \end{aligned}$$

Note that $L \in lfp(P)$ if and only if $lfp^{\#}(P)(L) > 0$ by definition. The next corollary shows that multiplicities of literals in least fixed points can be computed efficiently.

Corollary 4. *For every signature Γ and ground Datalog program P over Γ, a representation of the least fixed point with multiplicities $lfp^{\#}(P)$ can be computed in linear time $O(|P|)$.*

We represent $lfp^{\#}(P)$ by its restriction to $lfp(P)$, i.e., by the relation $\{(L, lfp^{\#}(P)(L)) \mid L \in lfp(P)\}$ which contains all non-zero values of $lfp^{\#}(P)$. Thereby, we avoid enumerating the elements of the complement of the least fixed point $lfp(P)^c$.

Proof. The relation $\{(L, lfp^{\#}(P)(L)) \mid L \in lfp(P)\}$ can be computed from P and $lfp(P)$ in time $O(|P|)$, by inspecting all clauses of P exactly once, and counting for all literals how often they appear on the left-hand side of clauses whose literals on the right-hand side all belong to $lfp(P)$. It is thus sufficient to compute the set $lfp(P)$ in time $O(|P|)$. This can be done by Theorem 3. $\square$

3.2. Characterization of inclusion

Let A be a tree automaton over Σ. We call a state $p \in sta(A)$ *accessible* (or sometimes *reachable*) if there exists a tree $t \in T_\Sigma$ such that $p \in eval_A(t)$, and *co-accessible* if there exists a tree $C[p] \in T_{\Sigma \cup \{p\}}$ with a unique occurrence of p (i.e., a context with hole marker p) such that $eval_A(C[p]) \cap fin(A) \neq \emptyset$. For every term $s \in T_\Sigma$, we denote by $C[s] \in T_\Sigma$ the term obtained by replacing the unique occurrence of p in $C[p]$ by s.

We call A *productive* if all its states are accessible and co-accessible. Note that productive automata may become unproductive by completion, since sink states are not co-accessible. The ground Datalog program in Fig. 3 computes all accessible and co-accessbile states of an automaton A. The rules of A are transformed to clauses of the Datalog program. The overall number of such clauses is linear in the size of A, so the least fixed point can be computed in linear time by Theorem 3. States that are unaccessible or not co-accessible, and all rules using them can be safely removed from A. This renders A productive in linear time, while preserving its language.

We will use the following notion for stepwise tree automata A with states $p_1, p_2 \in sta(A)$, meaning that evaluation may proceed in this pair:

$$A \models p_1 @ p_2 \Leftrightarrow_{df} \exists p \in sta(A) \cdot p_1 @ p_1 \to p \in rul(A)$$

Let B be another automaton over Σ but without ϵ-rules. The *product* $A \times B$ has state set $sta(A) \times sta(B)$, and rules inferred as follows:

$$\frac{a \to p \in rul(A) \quad a \to q \in rul(B)}{a \to (p, q)} \qquad \frac{p_1 @ p_2 \to p \in rul(A) \quad q_1 @ q_2 \to q \in rul(B)}{(p_1, q_1) @ (p_2, q_2) \to (p, q)} \qquad \frac{p' \xrightarrow{\epsilon} p \in rul(A) \quad q \in sta(B)}{(p', q) \xrightarrow{\epsilon} (p, q)}$$

We do not care about final states of $A \times B$ since these are useless in our characterization of inclusion. The following property of states p of A and q of B is equivalent to accessibility of the pair (p, q) in $A \times B$:

$$A, B \models \mathsf{acc}(p, q) \Leftrightarrow_{df} (p, q) \text{ accessible in } A \times B$$

$$\frac{a \to q \in \mathrm{rul}(A)}{\mathsf{acc}(q).} \qquad \frac{q_1 @ q_2 \to q \in \mathrm{rul}(A)}{\mathsf{acc}(q) \coloneq \mathsf{acc}(q_1), \mathsf{acc}(q_2).} \qquad \frac{q \xrightarrow{\epsilon} q'}{\mathsf{acc}(q') \coloneq \mathsf{acc}(q).}$$

$$\frac{q \in \mathrm{fin}(A)}{\mathsf{coacc}(q).} \qquad \frac{q_1 @ q_2 \to q \in \mathrm{rul}(A)}{\begin{array}{c}\mathsf{coacc}(q_1) \coloneq \mathsf{coacc}(q), \mathsf{acc}(q_2). \\ \mathsf{coacc}(q_2) \coloneq \mathsf{coacc}(q), \mathsf{acc}(q_1).\end{array}} \qquad \frac{q \xrightarrow{\epsilon} q'}{\mathsf{coacc}(q) \coloneq \mathsf{coacc}(q').}$$

Fig. 3. Accessible and co-accessible states of A.

Language inclusion $L(A) \subseteq L(B)$ fails under the following three conditions:
$A, B \models \texttt{fail}_0$: there exists a rule $a \to p \in rul(A)$ but no state $q \in sta(B)$ such that $a \to q \in rul(B)$;
$A, B \models \texttt{fail}_1$: there exist states p_1, p_2, q_1, q_2 such that $A, B \models \mathsf{acc}(p_1, q_1)$, $A, B \models \mathsf{acc}(p_2, q_2)$, $A \models p_1 \underline{@} p_2$ and $B \not\models q_1 \underline{@} q_2$;
$A, B \models \texttt{fail}_2$: there exist $p \in fin(A)$ and $q \notin fin(B)$ such that $A, B \models \mathsf{acc}(p, q)$.
We compose the properties of automata pairs by first-order connectives: we write $A, B \models \phi_1 \vee \phi_2$ iff $A, B \models \phi_1$ or $A, B \models \phi_2$, and similarly for the other first-order connectives such as $A, B \models \phi \Rightarrow \phi'$.

Proposition 5. *Inclusion $L(A) \subseteq L(B)$ for productive stepwise tree automata A with ϵ-rules and deterministic stepwise tree automata B fails iff $A, B \models \texttt{fail}_0 \vee \texttt{fail}_1 \vee \texttt{fail}_2$.*

Proof. For soundness, we suppose that one of the failure conditions holds, and show that some tree $t \in L(A)$ witnesses inclusion failure, i.e., $t \notin L(B)$.
$A, B \models \texttt{fail}_0$**.** Let us consider a rule $a \to p \in rul(A)$ such that no rule $a \to q \in rul(B)$ exists. Since A is productive, state p is co-accessible, i.e., there exists a term $C[p] \in T_{\Sigma \cup \{p\}}$ with a single occurrence of p such that $eval_A(C[p]) \cap fin(A) \neq \emptyset$. Hence $C[a] \in L(A)$. But $C[a] \notin L(B)$ because there is no rule $a \to q \in rul(B)$.
$A, B \models \texttt{fail}_1$**.** There exists $t_1 \in T_\Sigma$ such that $(p_1, q_1) \in eval_{A \times B}(t_1)$ by accessibility of (p_1, q_1) and there exists $t_2 \in T_\Sigma$ such that $(p_2, q_2) \in eval_{A \times B}(t_2)$ by accessibility of (p_2, q_2). Since $p_1 @ p_2 \to p \in rul(A)$ we also get $p \in eval_A(t_1 @ t_2)$ by definition of $eval_A$. Furthermore since A is productive there exists a term $C[p] \in T_{\Sigma \cup \{p\}}$ with a single occurrence p such that $C[t_1 @ t_2] \in L(A)$. Since B is deterministic it follows that $q_1 \in eval_B(t_1)$ and $q_2 \in eval_B(t_2)$ are unique. By hypothesis there is no q such that $q_1 @ q_2 \to q \in rul(B)$, so that $C[t_1 @ t_2] \notin L(B)$.
$A, B \models \texttt{fail}_2$**.** There are $p \in fin(A)$ and $q \notin fin(B)$ such that (p, q) is accessible. Thus, there exists $t \in T_\Sigma$ such that $(p, q) \in eval_{A \times B}(t)$. The state p is final in A, hence $t \in L(A)$. Since B is deterministic $q \in eval_B(t)$ is unique but q not final in B implies $t \notin L(B)$.
For completeness, we assume that there exists a tree $t \in L(A)$ such that $t \notin L(B)$, and show that some failure condition holds. There are two cases to be considered, depending on $eval_B(t)$.

(i) Assume $eval_B(t) = \emptyset$. There exists a minimal subtree t' of t such that $eval_B(t') = \emptyset$, too. If $t' = a$ is a leaf then $eval_A(a) \neq \emptyset$, since $t \in L(A)$, and $eval_B(a) = \emptyset$, hence $A, B \models \texttt{fail}_0$. If $t' = t_1 @ t_2$, then there exist $p_1 \in eval_A(t_1)$, $p_2 \in eval_A(t_2)$ and $p_1 @ p_2 \to p \in rul(A)$, since $t \in L(A)$. Since t' is defined as a minimal subtree and B is deterministic, $eval_B(t_1) = \{q_1\}$, $eval_B(t_2) = \{q_2\}$, and since $eval_B(t') = \emptyset$, there is no rule $q_1 @ q_2 \to q \in rul(B)$. This shows $A, B \models \texttt{fail}_1$.
(ii) If $eval_B(t) \neq \emptyset$ then there exists $q \in eval_B(t)$; since B is deterministic, q is necessarily unique. Since $t \notin L(B)$ this yields $q \notin fin(B)$. Moreover, since $t \in L(A)$, there exists $p \in eval_A(t) \cap fin(A)$. Thus, $A, B \models \texttt{fail}_2$ holds. $\square$

3.3. Testing characterization in ground Datalog

We next transform stepwise tree automaton A and B into a ground Datalog program by which to test the failure conditions.
Fig. 4 presents the transformation of two automata A, B into a ground Datalog program $D_0(A, B)$, which tests whether $A, B \models \texttt{fail}_0$ or $A, B \models \texttt{fail}_2$. The clauses produced from A and B by three transformation rules $(\mathsf{acc}_{/1})$, $(\mathsf{acc}_{/2})$, and $(\mathsf{acc}_{/3})$ compute the accessibility relation acc of $A \times B$ as usual. Clearly, $\mathsf{acc}(p, q) \in lfp(D_0(A, B))$ iff $A, B \models \mathsf{acc}(p, q)$. The Datalog clauses produced by transformation rules (fail_0) and (fail_2) serve for computing the predicates $\texttt{fail}_0$ and $\texttt{fail}_2$. By construction, $\texttt{fail}_0 \in lfp(D_0(A, B))$ iff $A, B \models \texttt{fail}_0$ and $\texttt{fail}_2 \in lfp(D_0(A, B))$ iff $A, B \models \texttt{fail}_2$.

$$(\mathsf{acc}_{/1})\ \frac{a \to p \in \mathrm{rul}(A) \qquad a \to q \in \mathrm{rul}(B)}{\mathsf{acc}(p, q).}$$

$$(\mathsf{acc}_{/2})\ \frac{p' \xrightarrow{\epsilon}_A p \in \mathrm{rul}(A) \qquad q \in \mathrm{sta}(B)}{\mathsf{acc}(p, q) :\!-\ \mathsf{acc}(p', q).}$$

$$(\mathsf{acc}_{/3})\ \frac{p_1 @ p_2 \to p \in \mathrm{rul}(A) \qquad q_1 @ q_2 \to q \in \mathrm{rul}(B)}{\mathsf{acc}(p, q) :\!-\ \mathsf{acc}(p_1, q_1), \mathsf{acc}(p_2, q_2).}$$

$$(\mathrm{fail}_0)\ \frac{a \to p \in \mathrm{rul}(A) \qquad \nexists q \in \mathrm{sta}(B).\ a \to q \in \mathrm{rul}(B)}{\mathrm{fail}_0.}$$

$$(\mathrm{fail}_2)\ \frac{p \in \mathrm{fin}(A) \qquad q \notin \mathrm{fin}(B)}{\mathrm{fail}_2 :\!-\ \mathsf{acc}(p, q).}$$

Fig. 4. Datalog program $D_0(A, B)$ testing $A, B \models \texttt{fail}_0$ and $A, B \models \texttt{fail}_2$.

$$(\mathsf{frb}_{/1})\ \frac{A \models p_1 \underline{@} p_2 \qquad B \not\models q_1 \underline{@} q_2}{\mathsf{frb}(p_2, q_2) :\!-\ \mathsf{acc}(p_1, q_1).}$$

$$(\mathsf{frb}_{/2})\ \frac{A \models p_1 \underline{@} p_2 \qquad B \not\models q_1 \underline{@} q_2}{\mathsf{frb}(p_1, q_1) :\!-\ \mathsf{acc}(p_2, q_2).}$$

$$(\mathrm{fail}_1)\ \frac{p \in \mathrm{sta}(A) \qquad q \in \mathrm{sta}(B)}{\mathrm{fail}_1 :\!-\ \mathsf{acc}(p, q), \mathsf{frb}(p, q)}$$

Fig. 5. Datalog program $D_1(A, B)$ testing $A, B \models \mathtt{fail}_1$.

Datalog program $D_0(A, B)$ can be computed in combined linear time $O(|A| \cdot |B|)$ from automata A and B, so that its size is in $O(|A| \cdot |B|)$. Furthermore, we can compute the least fixed point $lfp(D_0(A, B))$ in combined linear time, too (Theorem 3).

Whether $A, B \models \mathtt{fail}_1$ can be tested in $O(|A| \cdot |B|)$ is non-trivial though. To this purpose, we introduce a binary predicate $\mathsf{frb}(p, q)$ of *forbidden states*, which is equivalent to the implication $\mathsf{acc}(p, q) \rightarrow \mathtt{fail}_1$, i.e.:

$$A, B \models \mathsf{frb}(p, q) \Leftrightarrow_{df} A, B \models \mathsf{acc}(p, q) \Rightarrow \mathtt{fail}_1.$$

Inclusion is thus violated if forbidden states are accessible. The following lemma is an immediate consequence of the definitions.

Lemma 6. *$A, B \models \mathsf{frb}(p, q)$ iff there are p', q' such that one of the following two conditions holds*:

1. $A \models p \underline{@} p' \wedge A, \quad B \models \mathsf{acc}(p', q') \wedge B \not\models q \underline{@} q'$, *or*
2. $A \models p' \underline{@} p \wedge A, \quad B \models \mathsf{acc}(p', q') \wedge B \not\models q' \underline{@} q$.

Fig. 5 extends $D_0(A, B)$ to $D_1(A, B)$ by clauses for $\mathtt{fail}_1$. These are produced by transformation rules ($\mathrm{frb}_{/1}$) and ($\mathrm{frb}_{/2}$), that are justified by Lemma 6. Transformation rule (fail_1) is witnessed by the definition of $A, B \models \mathsf{frb}(p, q)$. It should be clear that $A, B \models \mathtt{fail}_1$ iff $\mathtt{fail}_1 \in lfp(D_1(A, B))$.

Proposition 7. *Let A and B be stepwise tree automata for binary trees. If A is productive and B deterministic then*:

$$L(A) \subseteq L(B) \Leftrightarrow lfp(D_1(A, B)) \cap \{\mathtt{fail}_0, \mathtt{fail}_1, \mathtt{fail}_2\} = \emptyset$$

Proof. From Proposition 5 and Lemma 6. □

However, the number of clauses produced by transformation rules ($\mathrm{frb}_{/1}$) and ($\mathrm{frb}_{/2}$) may sum up to $O(|A| \cdot |sta(B)|^2)$ in the worst case. The overall size of the Datalog program $D_1(A, B)$ is thus bounded by $O(|A| \cdot |B|^2)$. Programs of this size may arise, as shown by the example in Fig. 6. Even though in this case $O(|A| \cdot |B|) = O(n)$, there are n^2 clauses in $D_1(A, B)$ produced by transformation rule ($\mathrm{frb}_{/2}$).

By computing the least fixed point of $D_1(A, B)$, we can thus decide language inclusion $L(A) \subseteq L(B)$ in time $O(|A| \cdot |B|^2)$. Unfortunately, this is not yet any better than the naive algorithm through complementation discussed in Section 1.

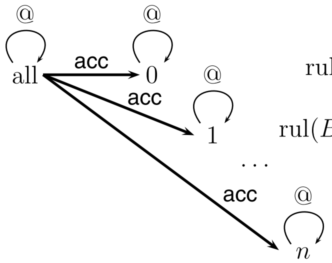
3.4. Inclusion test in time $O(|A| \cdot |B|)$

The problem with the clauses produced by ($\mathrm{frb}_{/1}$) and ($\mathrm{frb}_{/2}$) is the enumeration of clauses for forbidden states. This operation makes negative information of B explicit that one would like to leave implicit.

To see this, let $\Sigma = \{a\}$ and let us consider the example in Fig. 6. There, automaton A has a unique (final) state such that $sta(A) = fin(A) = \{\text{all}\}$. It recognizes all binary trees over $\Sigma_@$ whose paths to the left are of arbitrary length. They can be obtained as Curried encodings of unranked trees whose nodes have arbitrary many children. Automaton $B_{\leq n}$ has states $sta(B_{\leq n}) = fin(B_{\leq n}) = \{0, \ldots, n\}$. It recognizes the set of binary trees over $\Sigma_@$ whose paths to the left are bounded by n. Thus, their Curried encodings cannot have more than n children. Note that the rules such that $A \models p \underline{@} p$, resp. $B_{\leq n} \models q \underline{@} q$, are depicted in Fig. 6 by @-loops on states $p \in sta(A)$, resp. $q \in sta(B_{\leq n})$. Accessibility $A, B_{\leq n} \models \mathsf{acc}(\text{all}, j)$ holds for all $0 \leq j \leq n$ and implies forbidden states $A, B_{\leq n} \models \mathsf{frb}(\text{all}, i)$ for all $0 \leq i \leq n$. This is inferred by quadratically many clauses $\mathsf{frb}(\text{all}, i) :\!-\ \mathsf{acc}(\text{all}, j)$ that need to be avoided.

The idea is to count positive information in order to deduce how many times negative information can be inferred. Given a state p, we count the number of pairs (p', q') with $A, B \models \mathsf{acc}(p', q')$ and $A \models p' \underline{@} p$ or *vice versa*, and compare it with the number of such pairs that raise $\mathsf{frb}(p, q)$:

$$l(p) = \#\{(p', q') \mid A \models p' \underline{@} p \wedge A, B \models \mathsf{acc}(p', q')\} + \#\{(p', q') \mid A \models p \underline{@} p' \wedge A, B \models \mathsf{acc}(p', q')\}$$
$$l(p, q) = \#\{(p', q') \mid A \models p' \underline{@} p \wedge A, B \models \mathsf{acc}(p', q') \wedge B \models q' \underline{@} q\} + \#\{(p', q') \mid A \models p \underline{@} p' \wedge A, B \models \mathsf{acc}(p', q') \wedge B \models q \underline{@} q'\}$$



$$\begin{aligned} \mathrm{rul}(A) = \{\ & a \to \mathrm{all}, \\ & \mathrm{all}@\mathrm{all} \to \mathrm{all}\ \} \\ \mathrm{rul}(B_{\le n}) = \{\ & a \to 0 \\ & 0@0 \to 1, \\ & \ldots \\ & n{-}1@n{-}1 \to n\ \} \end{aligned}$$

counters:
$l(\mathrm{all}) = n{+}1$
$l(\mathrm{all}, 0) = 1$
$\ldots$
$l(\mathrm{all}, n) = 1$

Fig. 6. Rule $(\mathsf{frb}_{/2})$ can infer, for all $0 \le i \ne j \le n$, the clause $\mathsf{frb}(\mathrm{all}, i) :- \ \mathsf{acc}(\mathrm{all}, j)$.

for all $p \in \mathrm{sta}(A)$ do $l(p) := 0$;
  for all $q \in \mathrm{sta}(B)$ do $l(p, q) := 0$;
for all $p_1@p_2 \to p$ in $\mathrm{rul}(A)$ do
  for all $q \in \mathrm{sta}(B)$ do
    if $\mathsf{acc}(p_1, q) \in \mathrm{lfp}(D_0(A, B))$ then increment $l(p_2)$;
    if $\mathsf{acc}(p_2, q) \in \mathrm{lfp}(D_0(A, B))$ then increment $l(p_1)$;
  for all $q_1@q_2 \to q$ in $\mathrm{rul}(A)$ do
    if $\mathsf{acc}(p_1, q_1) \in \mathrm{lfp}(D_0(A, B))$ then increment $l(p_2, q_2)$;
    if $\mathsf{acc}(p_2, q_2) \in \mathrm{lfp}(D_0(A, B))$ then increment $l(p_1, q_1)$;

Fig. 7. Counting in $O(|A| \cdot |B|)$.

Lemma 8. *$A, B \models \mathsf{frb}(p, q)$ iff $l(p) > l(p, q)$.*

Proof. By definition, $l(p) \ge l(p, q)$ for all p, q. We have $l(p) > l(p, q)$ iff there are p', q' such that $A \models p'\underline{@}p \wedge A, B \models \mathsf{acc}(p', q') \wedge B \not\models q'\underline{@}q$ or symmetrically $A \models p\underline{@}p' \wedge A, B \models \mathsf{acc}(p', q') \wedge B \not\models q\underline{@}q'$. By Lemma 6, this is equivalent to $A, B \models \mathsf{frb}(p, q)$. $\square$

It remains to see that we can compute the collection of numbers $l(p)$ and $l(p, q)$ for all $p \in sta(A)$ and $q \in sta(B)$ in time $O(|A| \cdot |B|)$. This can be done by the algorithm in Fig. 7.

Theorem 9. *Let A and B be tree automata over a ranked signature Σ, possibly with ϵ-rules in A. If B is deterministic, then inclusion $L(A) \subseteq L(B)$ can be decided in time $O(|A| \cdot |B|)$ independently of the size of Σ.*

Proof. We can assume that A and B are stepwise tree automata by Proposition 1. We first compute $D_0(A, B)$ from A and B in combined linear time $O(|A| \cdot |B|)$, and then the least fixed point of this Datalog program in the same time. If $lfp(D_0(A, B))$ contains $\mathtt{fail}_0$ or $\mathtt{fail}_2$, then inclusion $L(A) \subseteq L(B)$ does not hold. Otherwise, we compute all numbers $l(p)$ and $l(p, q)$ in time $O(|A| \cdot |B|)$ by the algorithm in Fig. 7, and test for all $\mathsf{acc}(p, q) \in lfp(D_0(A, B))$ whether $A, B \models \mathsf{frb}(p, q)$. Inclusion $L(A) \subseteq L(B)$ holds iff this test succeeds. It can be performed in time $O(|A| \cdot |B|)$ by checking the values of the counters (Lemma 8). $\square$

3.5. Efficient algorithm

The previous algorithm has a satisfactory worst case complexity of $O(|A| \cdot |B|)$. In practice, however, it is non-optimal with respect to average time efficiency.

The first problem is that all pairs of rules in A and B are enumerated when computing the values of the counters. We now present a better algorithm, which inspects at most the accessible part of $A \times B$. The second problem is that (fail_1) is applied only after the fixed point computation. From now on, we envisage an algorithm (presented in full in Section 3.6) that detects inclusion failure as early as possible so that we do not have to complete the fixed point computation in such cases. These cases are very frequent in practice, as we will show experimentally (in Section 6), so that the gain in efficiency is considerable.

$$(\mathsf{frb}^c_{/1})\ \frac{A \models p_1\underline{@}p_2 \qquad q_2 \in \mathrm{sta}(B)}{\mathsf{frb}^c(p_1, Q^B_1(q_2)) \coloneq \mathsf{acc}(p_2, q_2).} \quad Q^B_1(q_2) = \{q_1 \mid B \models q_1\underline{@}q_2\}$$

$$(\mathsf{frb}^c_{/2})\ \frac{A \models p_1\underline{@}p_2 \qquad q_1 \in \mathrm{sta}(B)}{\mathsf{frb}^c(p_2, Q^B_2(q_1)) \coloneq \mathsf{acc}(p_1, q_1).} \quad Q^B_2(q_1) = \{q_2 \mid B \models q_1\underline{@}q_2\}$$

Fig. 8. Grouping clauses from $(\mathrm{frb}_{/1})$ and $(\mathrm{frb}_{/2})$.

$$\frac{\begin{matrix} p_1@p_2 \to p \in \mathrm{rul}(A) \\ q_1 \in \mathrm{sta}(B) \end{matrix} \qquad \left.\begin{matrix} q_1@q_2^1 \to q^1 \in \mathrm{rul}(B) \\ \vdots \\ q_1@q_2^n \to q^n \in \mathrm{rul}(B) \end{matrix}\right\} \begin{matrix}\text{all rules} \\ \text{for } q_1\end{matrix}}{\begin{matrix} \mathsf{frb}^c(p_2, \{q_2^1, \ldots, q_2^n\}) \coloneq \mathsf{acc}(p_1, q_1). \\ \mathsf{acc}(p, q^1) \coloneq \mathsf{acc}(p_1, q_1), \mathsf{acc}(p_2, q_2^1). \\ \vdots \\ \mathsf{acc}(p, q^n) \coloneq \mathsf{acc}(p_1, q_1), \mathsf{acc}(p_2, q_2^n). \end{matrix}}$$

Fig. 9. Rewriting groups of $(\mathrm{frb}_{/2})$ clauses to $(\mathrm{frb}^c_{/2})$ clauses.

We introduce literals $\mathsf{frb}^c(p, Q)$ for states $p \in sta(A)$ and state sets $Q \subseteq sta(B)$ with the following semantics:

$$A, B \models \mathsf{frb}^c(p, Q) \Leftrightarrow \forall q \in sta(B)\backslash Q.\ A, B \models \mathsf{frb}(p, q)$$

In Fig. 6, we have $A, B \models \mathsf{frb}^c(0, \{i\})$ for all $1 \le i \le n$. Thus, all literals $\mathsf{frb}(0, i)$ are implied by $n-1$ literals $\mathsf{frb}^c(0, \{j\})$. This multiplicity $n-1$ is equal to $l(0) - l(0, i)$. Indeed, our objective is to compute the set of all literals satisfying $A, B \models \mathsf{frb}^c(p, Q)$ by a Datalog program and to infer the values of $l(p) - l(p, q)$ thereby.

Note that two literals $\mathsf{frb}^c(p, Q)$ and $\mathsf{frb}^c(p, Q')$ are equal syntactically if and only if $Q = Q'$. In order to make this happen technically, we assume a fixed total order $<$ on $sta(B)$ in order to identify $\mathsf{frb}^c(p, Q)$ with the unique $(n+1)$-ary literal $\mathsf{frb}^c(p, q_1, \ldots, q_n)$ with $Q = \{q_1, \ldots, q_n\}$ and $q_1 < \ldots < q_n$. Thereby, all results on ground Datalog programs continue to apply.

In Fig. 8, we present transformation rules $(\mathrm{frb}^c_{/1})$ and $(\mathrm{frb}^c_{/2})$, which produce Datalog clauses inferring $\mathsf{frb}^c(p, Q)$ literals. They group many clauses produced by a transformation rule $(\mathrm{frb}_{/i})$. Consider $i = 2$. The transformation rule assumes $A \models p_1\underline{@}p_2$ and a state $q_1 \in sta(B)$. It then computes the set $Q^B_2(q_1)$ of all states q_2 with $B \models q_1\underline{@}q_2$, and produces the clause $\mathsf{frb}^c(p_2, Q^B_2(q_1)) \coloneq \mathsf{acc}(p_1, q_1)$. This is correct, since if $A, B \models \mathsf{acc}(p_1, q_1)$, then for all $q_2 \notin Q^B_2(q_1)$, we have $A, B \models \mathsf{frb}(p_2, q_2)$ and thus $A, B \models \mathsf{frb}^c(p_2, Q^B_2(q_1))$. In Fig. 6, for instance, transformation $(\mathrm{frb}^c_{/1})$ produces for all $1 \le i \le n$ the clauses $\mathsf{frb}^c(0, \{i\}) \coloneq \mathsf{acc}(0, i)$. The overall size of these clauses is linear in n, no more quadratic!

Let $D_2(A, B)$ be the ground Datalog program which extends $D_0(A, B)$ by the clauses from $(\mathrm{frb}^c_{/1})$ and $(\mathrm{frb}^c_{/2})$. This program remains incomplete, in that $\mathsf{frb}^c(p, Q)$ literals are never used in order to infer $\mathtt{fail}_1$.

Lemma 10. *$D_2(A, B)$ can be computed in time $O(|A| \cdot |B|)$ from A and B.*

Proof. We have seen the result for $D_0(A, B)$ already. The number of clauses produced by transformation rule $(\mathrm{frb}^c_{/1})$ is in $O(|A| \cdot |sta(B)|)$ but the size of each such clause is $n + 1$ which in the worst case could be $|sta(B)| + 1$. Symmetrically for $(\mathrm{frb}^c_{/2})$. The overall size of all frb^c clauses, however, is bounded by the overall number of acc clauses produced at the same time, which in turn is bounded by $O(|A| \cdot |B|)$, too! To see this, we can rewrite the first rule of $(\mathrm{frb}^c_{/2})$ as shown in Fig. 9, such that the corresponding $(\mathrm{acc}_{/3})$ clauses are inferred simultaneously (and these do not overlap).

It remains to show how to compute $D_2(A, B)$ in combined linear time. The following program produces all clauses from transformation rule $(\mathrm{frb}^c_{/2})$:

for all $p_1@p_2 \to p \in rul(A)$ do
  for all $q_1 \in sta(B)$ do
    compute $Q = Q^B_2(q_1)$;
    collect $\mathsf{frb}^c(p_2, Q) \coloneq \mathsf{acc}(p_1, q_1)$;

The set Q can be computed in time $O(|Q|)$ from a precomputed data structure that returns for a given state q_1 all rules $q_1@q_2 \to q$ in $rul(B)$ in linear time depending on their number. The whole programs thus runs in time $O(|D_2(A, B)|)$ which is in $O(|A| \cdot |B|)$. $\square$

The Datalog programs $D_1(A,B)$ and $D_2(A,B)$ have the same clauses for literals $\mathsf{acc}(p,q)$, $\mathtt{fail}_0$, and $\mathtt{fail}_2$, so their least fixed points coincide for these. In particular, we can decide $A,B \models \mathtt{fail}_0 \vee \mathtt{fail}_2$ in time $O(|A| \cdot |B|)$ by testing membership of $\mathtt{fail}_0$ and $\mathtt{fail}_2$ in $lfp(D_2(A,B))$. It remains to relate both programs with respect to forbidden states and $\mathtt{fail}_1$.

Lemma 11. *A literal* $\mathsf{frb}(p,q)$ *belongs to* $lfp(D_1(A,B))$ *if and only if there exists a set* $Q \subseteq sta(B)$ *not containing* q *such that* $\mathsf{frb}^c(p,Q) \in lfp(D_2(A,B))$.

Proof. Suppose that $\mathsf{frb}(p_1,q_1) \in lfp(D_1(A,B))$. The previous literal has been added by a clause produced by $(\mathrm{frb}_{/1})$ or $(\mathrm{frb}_{/2})$. By symmetry it is sufficient to consider the first case only. The contributing clause of $D_1(A,B)$ must be of the form $\mathsf{frb}(p_1,q_1) \coloneq \mathsf{acc}(p_2,q_2)$. $(\mathrm{frb}_{/1})$ assumes $A \models p_1 \underline{@} p_2$ and $B \not\models q_1 \underline{@} q_2$, so that $q_1 \notin Q_1(q_2)$. $(\mathrm{frb}^c_{/1})$ produces the clause $\mathsf{frb}^c(p_1, Q_1^B(q_2)) \coloneq \mathsf{acc}(p_2,q_2)$ in $D_2(A,B)$. Since $\mathsf{acc}(p_2,q_2) \in lfp(D_1(A,B))$, we equally have $\mathsf{acc}(p_2,q_2) \in lfp(D_2(A,B))$. Hence, the above clause of $D_2(A,B)$ is applicable and adds $\mathsf{frb}^c(p_1, Q_1^B(q_2))$ to $lfp(D_2(A,B))$. The inverse argument is similar. $\square$

Lemma 12. *For* $D = D_2(A,B)$, $p \in sta(A)$ *and* $q \in sta(B)$:

$$\begin{array}{rcl} l(p) & = & \sum_{Q \subseteq sta(B)} lfp^{\#}(D)(\mathsf{frb}^c(p,Q)) \\ l(p,q) & = & \sum_{Q \subseteq sta(B), q \in Q} lfp^{\#}(D)(\mathsf{frb}^c(p,Q)) \end{array}$$

Proof. This follows from the definitions, as we elaborate in the first case:

$$\begin{aligned} l(p) &= \#\{(p',q') \mid A \models p' \underline{@} p \wedge A,B \models \mathsf{acc}(p',q')\} + \#\{(p',q') \mid A \models p \underline{@} p' \wedge A,B \models \mathsf{acc}(p',q')\} \\ &= \#\{(p',q') \mid \mathsf{frb}^c(p, Q_2^B(q')) \coloneq \mathsf{acc}(p',q').\ \in D \wedge \mathsf{acc}(p',q') \in lfp(D)\} \\ &\quad + \#\{(p',q') \mid \mathsf{frb}^c(p, Q_1^B(q')) \coloneq \mathsf{acc}(p',q').\ \in D \wedge \mathsf{acc}(p',q') \in lfp(D)\} \\ &= \sum_{Q \subseteq sta(B)} lfp^{\#}(D)(\mathsf{frb}^c(p,Q)) \qquad \square \end{aligned}$$

We can thus compute all numbers $l(p)$ and $l(p,q)$ in linear time depending on the size of $lfp(D_2(A,B))$ by Corollary 4.

Inclusion $L(A) \subseteq L(B)$ can be tested as before, except that the numbers $l(p) - l(p,q)$ can now be computed from $lfp(D_2(A,B))$ more efficiently. This requires computing the accessible part of $A \times B$ only, by lazily creating only the needed clauses from $(\mathrm{acc}_{/3})$ as usual. The application of all rules $(\mathrm{frb}^c_{/1})$ and $(\mathrm{frb}^c_{/2})$ can be done in time $O(|A| \cdot |sta(B)| + |rul(B)|)$, which may be much smaller than $O(|A \times B|)$, too.

3.6. Early failure detection

We next tackle the problem that (fail_1) is tested only after fixed point computation at the end, by the following loop:

for all $\mathsf{acc}(p,q) \in lfp(D_2(A,B))$ do
  if $l(p) > l(p,q)$ then return false;
otherwise return true;

Instead, we approach on the fly checking for (fail_1) as follows. Once a literal $\mathsf{acc}(p,q)$ is inferred during the computation of the fixed point of $D_2(A,B)$, we check in constant time whether the current values of the counters satisfy $l(p) > l(p,q)$, i.e., whether $\mathsf{frb}(p,q)$ is implied by some literal $\mathsf{frb}^c(p,Q)$ inferred before with $q \notin Q$. This requires updating the counters on the fly, but this is not difficult if we update them with priority.

The main difficulty arises when deriving $\mathsf{frb}^c(p,Q)$ only after some $\mathsf{acc}(p,q)$, since we cannot check for all $q \in sta(B) \backslash Q$ whether $\mathsf{acc}(p,q)$ has been inferred before without enumerating the complement of Q. It turns out fortunately that all tests for (fail_1) come for free and on the fly (without any testing after fixed point computation) if we impose the following *priority* discipline. We assume that literals of the form $\mathsf{acc}(p,q)$ are always inferred with the lowest priority, i.e., whenever other literals can be inferred at the same time, these will be inferred before.

Our *on-the-fly algorithm* thus computes the least fixed point of $D_2(A,F)$ with the above priorities. The counters $l(p)$ and $l(p,q)$ are always updated immediately. Whenever a literal $\mathsf{acc}(p,q)$ is inferred, the counters are tested for $l(p) > l(p,q)$. If this test succeeds, the algorithm returns false, otherwise it continues and returns true at the very end.

Lemma 13. *The on-the-fly algorithm correctly detects* (fail_1) *if a literal* $\mathsf{acc}(p_1,q_1)$ *is inferred before some literal* $\mathsf{frb}^c(p_1,Q_1)$ *with* $q_1 \notin Q_1$.

Proof. This situation is depicted in Fig. 10. Literal $\mathsf{frb}^c(p_1,Q_1)$ originates from a clause produced by rule (frb^c) and some literal $\mathsf{acc}(p_2,q_2)$ added earlier:

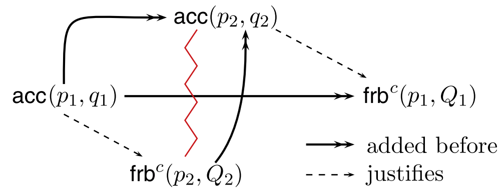


Fig. 10. Early failure detection: $\mathsf{frb}^c(p_2, q_2)$ in $lfp(D_2(A, B))$ before $\mathsf{acc}(p_2, q_2)$.

$$\frac{p_1@p_2 \to p \in rul(A) \qquad Q_1 = Q_1^B(q_2)}{\mathsf{frb}^c(p_1, Q_1) :\!-\ \mathsf{acc}(p_2, q_2).}$$

We show by contradiction that $\mathsf{acc}(p_1, q_1)$ has got added before $\mathsf{acc}(p_2, q_2)$. Otherwise, $\mathsf{acc}(p_2, q_2)$ has been added before $\mathsf{acc}(p_1, q_1)$, so that due to our priority assumption, $\mathsf{frb}^c(p_1, Q_1)$ has been added before $\mathsf{acc}(p_1, q_1)$ which contradicts the hypothesis. Having $\mathsf{acc}(p_1, q_1)$ in the fixed point permits to apply the following clause of (frb^c):

$$\frac{p_1@p_2 \to p \in rul(A) \qquad Q_2 = Q_2^B(q_1)}{\mathsf{frb}^c(p_2, Q_2) :\!-\ \mathsf{acc}(p_1, q_1).}$$

Note that $q_1 \in Q_1^B(q_2)$ iff $q_2 \in Q_2^B(q_1)$. Thus $q_2 \notin Q_2$ since $q_1 \notin Q_1$. Consequently, $\mathsf{acc}(p_2, q_2), \mathsf{frb}^c(p_2, Q_2) \in lfp(D_2(A, B))$ raises (fail_1), and this is correctly detected by the modified algorithm, since $\mathsf{frb}^c(p_2, Q_2)$ is inferred before $\mathsf{acc}(p_2, q_2)$. □

3.7. Incrementality

Incrementality appears to be critical for efficiency in our experiments. In our prime application to schema-guided query induction [9], for instance, we use incremental addition of ϵ-rules to the automaton A on the left. These model state merging operations $p_1 = p_2$ during automata induction as $p_1 \xrightarrow{\epsilon} p_2$ and $p_2 \xrightarrow{\epsilon} p_1$.

Fixed points of Datalog programs can be computed incrementally with respect to adding new clauses. Priorities, however, may raise trouble here. It would not be correct to add clauses later on, that should have been applied with priority before. In this case, one would have to redo some work.

Rules of automata A or B are transformed to clauses of $D_2(A, B)$. The incremental addition of ϵ-rules to A is harmless. They are transformed by ($\mathrm{acc}_{/2}$) to clauses with the lowest priority. All previous clauses remain valid (in contrast to adding rules $q_1@q_2 \to q$ to B which changes $Q_2^B(q_1)$). For these two reasons, we do not have to redo any work when adding ϵ-rules to A later on. Of course, incrementality assumes early failure detection.

4. Factorized tree automata

We next relax the determinism assumption on B in a controlled manner, that will be crucial to deal with DTDs. This leads us to introduce the notion of deterministic factorized tree automata, and to check inclusion for them. Inclusion in deterministic factorized automata is exactly what we need for inclusion in deterministic DTDs in Section 5.

4.1. Deterministic factorized tree automata

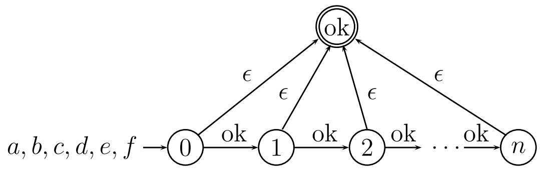
We replace B by deterministic factorized automata F, which we now introduce. These are stepwise tree automata with ϵ-rules for ranked trees, that represent deterministic stepwise tree automata in a more compact manner.

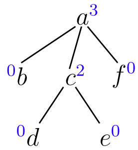
Definition 1. A *factorized tree automaton* F over a stepwise signature Σ consists of a stepwise tree automaton with ϵ-rules and a partition $sta(F) = sta_1(F) \uplus sta_2(F)$ such that for all $q_1@q_2 \to q$ in $rul(F)$ we have $q_1 \in sta_1(F)$ and $q_2 \in sta_2(F)$.

We say that q is of *sort i* in F if $q \in sta_i(F)$. The sort determines which states may be used in the ith position of the binary symbol @ in rules of F.

Every factorized automaton F defines a tree automaton $b(F)$ without ϵ-rules that recognizes the same language. Both automata have the same signature and states; the rules of $b(F)$ are inferred as follows from those of F:

$$(\mathrm{E}_1)\,\frac{a \to q \in rul(F)}{a \to q \in rul(b(F))} \qquad (\mathrm{E}_2)\,\frac{q_1 \xrightarrow{\epsilon}{}^{*}_{F} r_1 \quad q_2 \xrightarrow{\epsilon}{}^{*}_{F} r_2 \quad r_1@r_2 \to q \in rul(F)}{q_1@q_2 \to q \in rul(b(F))}$$





$$
\begin{aligned}
\mathrm{sta}_1(F_{\leq n}) &= \{0, \ldots, n\} \\
\mathrm{sta}_2(F_{\leq n}) &= \{\mathrm{ok}\} \\
\mathrm{rul}(F_{\leq n}) &= \{a, b, c, d, e, f \to 0 \xrightarrow{\epsilon} \mathrm{ok}, \\
& \quad 0@\mathrm{ok} \to 1 \xrightarrow{\epsilon} \mathrm{ok}, \\
& \quad \ldots, \\
& \quad n{-}1@\mathrm{ok} \to n \xrightarrow{\epsilon} \mathrm{ok}\}
\end{aligned}
$$

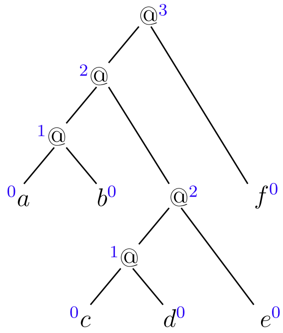


Fig. 11. On the left, we define for all n a deterministic factorized tree automaton $F_{\leq n}$. On the right, the lower tree is the Curried encoding of the upper. We annotate all nodes by the unique state assigned by the evaluator of the deterministic tree automaton $b(F_{\leq n})$ where $n \geq 3$.

We set $fin(b(F)) = \{q \mid q \xrightarrow{\epsilon}{}^*_F r,\ r \in fin(F)\}$. Note that the size of $b(F)$ may be $O(|rul(F)| \cdot |sta(F)|^2)$, which is cubic in that of F in the worst case. Besides their succinctness, the truly interesting bit about factorized tree automata is their notion of determinism.

A collection of examples for factorized tree automata $(F_{\leq n})_n$ is given in Fig. 11. The set of constants of $F_{\leq n}$ is $\Sigma = \{a, b, c, d, e, f\}$. Automaton $F_{\leq n}$ recognizes all binary trees over $\Sigma_@$ whose paths to the left are always of length at most n. These can be obtained as Curried encodings of unranked trees whose nodes have at most n children. The states of sort 1 of $F_{\leq n}$ are $\{0, \ldots, n\}$. For every node, they count the length of the path to the left-most leaf. The single state of sort 2 is ok. It can be assigned to all nodes rooting subtrees in the language of $F_{\leq n}$. Automaton $F_{\leq n}$ has rules $a, b, c, d, e, f \to 0$, rules $i{-}1@\mathrm{ok} \to i$ for all $1 \leq i \leq n$ and $i \xrightarrow{\epsilon} \mathrm{ok}$ for all $0 \leq i \leq n$. The size of $F_{\leq n}$ is thus in $O(n)$. The corresponding tree automaton $b(F_{\leq n})$ is of size $O(n^2)$, since it has rules $a, b, c, d, e, f \to 0$ and $i{-}1@j \to i$ for all $1 \leq i \leq n$ and $0 \leq j \leq n$. Note that $b(F_{\leq n})$ is the unique state-minimal deterministic automaton recognizing the language of $F_{\leq n}$. The sizes $|F_{\leq n}|$ are asymptotically smaller by a factor of n than $|b(F_{\leq n})|$. Nevertheless, all $F_{\leq n}$ are deterministic in the following sense.

Definition 2. A factorized tree automaton F is (bottom-up) *deterministic* if:

- $\mathtt{d}_0$: the ϵ-free part of F is (bottom-up) deterministic;
- $\mathtt{d}_1$: for all $q \in sta(F)$ and sorts $i \in \{1, 2\}$, there is at most one state r of sort i such that $q \xrightarrow{\epsilon}{}^*_F r$.

Non-redundant ϵ-rules must change the sort: if $q \xrightarrow{\epsilon}_F r$ for two states of the same sort, then $r = q$ by $\mathtt{d}_1$, and $q \xrightarrow{\epsilon}{}^*_F q$. A similar argument shows that all proper chains of ϵ-rules are redundant so that $\xrightarrow{\epsilon}{}^*_F$ is equal to $\xrightarrow{\epsilon}{}^{\leq 1}_F$.

Another consequence of determinism is that the size of deterministic $b(F)$ is at most quadratic in the size of deterministic F, since $b(F)$ cannot have more than $|sta(F)|^2$ many binary transitions.

Proposition 14. *The tree automaton $b(F)$ is deterministic for all deterministic factorized tree automata F.*

Proof. Let $B = b(F)$ which by construction is free of ϵ-rules. For every constant $a \in \Sigma$, the uniqueness of q such that $a \to q \in rul(B)$ follows from $\mathtt{d}_0$. For every $q_1@q_2 \to q$ in $rul(B)$ we have to show that q is uniquely determined by q_1 and q_2. By $\mathtt{d}_1$ there is at most one state r_1 of sort 1 such that $q_1 \xrightarrow{\epsilon}{}^*_F r_1$ and at most one state r_2 of sort 2 such that $q_2 \xrightarrow{\epsilon}{}^*_F r_2$. Condition $\mathtt{d}_0$ implies that there exists at most one state q such that $r_1@r_2 \to q \in rul(F)$. $\square$

Conversely, every deterministic stepwise tree automaton B can be converted in time $O(|B|)$ into a deterministic factorized tree automaton F, such that $b(F)$ is equal to B modulo state renaming. The states of F are $sta_i(F) = sta(B) \times \{i\}$ for $i \in \{1, 2\}$. Rules are transformed as follows:

$$(\mathsf{acc}_{/1a})\ \frac{a \to p \in \mathrm{rul}(A) \qquad a \to q \in \mathrm{rul}(F)}{\mathsf{acc}(p,q).}$$

$$(\mathsf{acc}_{/2a})\ \frac{p' \xrightarrow{\epsilon}_A p \in \mathrm{rul}(A) \qquad q \in \mathrm{sta}(F)}{\mathsf{acc}(p,q) :- \mathsf{acc}(p',q).}$$

$$(\mathsf{acc}_{/3a})\ \frac{p_1@p_2 \to p \in \mathrm{rul}(A) \qquad q_1@q_2 \to q \in \mathrm{rul}(F)}{\mathsf{acc}(p,q) :- \mathsf{f.acc}(p_1,q_1), \mathsf{f.acc}(p_2,q_2).}$$

$$(\mathsf{f.acc})\ \frac{p \in \mathrm{sta}(A) \qquad q \xrightarrow{\epsilon}{}^{\le 1}_F r}{\mathsf{f.acc}(p,r) :- \mathsf{acc}(p,q).}$$

$$(\mathrm{fail}_{0/a})\ \frac{a \to p \in \mathrm{rul}(A) \qquad \nexists q \in \mathrm{sta}(F).\ a \to q \in \mathrm{rul}(F)}{\mathrm{fail}_0.}$$

$$(\mathrm{fail}_{2/a})\ \frac{p \in \mathrm{fin}(A) \qquad \forall r \in \mathrm{sta}(F).\ q \xrightarrow{\epsilon}{}^{\le 1}_F r \Rightarrow r \notin \mathrm{fin}(F)}{\mathrm{fail}_2 :- \mathsf{acc}(p,q).}$$

Fig. 12. $D_0(A, F)$ testing $A, B \models \mathtt{fail}_0$ and $A, B \models \mathtt{fail}_2$ where $B = b(F)$.

$$\frac{q_1@q_2 \to q \in rul(B)}{\begin{array}{c}(q_1,1)@(q_2,2) \to (q,1) \in rul(F)\\ (q,1) \xrightarrow{\epsilon} (q,2) \in rul(F)\end{array}} \qquad \frac{a \to q \in rul(B)}{\begin{array}{c}a \to (q,1) \in rul(F)\\ (q,1) \xrightarrow{\epsilon} (q,2) \in rul(F)\end{array}}$$

4.2. Testing validity of $\mathtt{fail}_0$ and $\mathtt{fail}_2$

Given an automaton A and a deterministic factorized automaton F, we first characterize $A, b(F) \models \mathtt{fail}_0$ and $A, b(F) \models \mathtt{fail}_2$ in terms of A and F. This must be done without computing $b(F)$, since its size may be in $O(|sta(F)|^2)$.

Lemma 15. *Let $B = b(F)$.*

(1) $A, B \models \mathtt{fail}_0$ *iff* $\exists a \to p \in rul(A) \wedge \nexists a \to q \in rul(F)$

(2) $A, B \models \mathtt{fail}_2$ *iff* $\exists p \in fin(A)\ \exists q \in sta(F).\ A, B \models \mathsf{acc}(p,q) \wedge \forall r \in sta(F).\ q \xrightarrow{\epsilon}{}^{\le 1}_F r \Rightarrow r \notin fin(F)$

Proof. The first statement follows from construction rule (E_1) of $b(F)$. For the second, note that $q \notin fin(B)$ iff $\forall r \in sta(F).\ q \xrightarrow{\epsilon}{}^{\le 1}_F r \Rightarrow r \notin fin(F)$. Note that this universal quantifier is harmless, since for every q there is at most one r with $q \xrightarrow{\epsilon}{}^{\le 1}_F r$. We can now conclude straightforwardly:

$A, B \models \mathtt{fail}_2$

$\Leftrightarrow \exists p \in fin(A)\ \exists q \notin fin(B).\ A, B \models \mathsf{acc}(p,q)$

$\Leftrightarrow \exists p \in fin(A)\ \exists q \in sta(F).\ A, B \models \mathsf{acc}(p,q) \wedge \forall r.\ q \xrightarrow{\epsilon}{}^{\le 1}_F r \Rightarrow r \notin fin(F)$ □

There is a subtle difference between accessibility in F and $b(F)$: accessibility in $b(F)$ implies accessibility in F, but not *vice versa*. For instance, in Fig. 11, state ok is accessible in $F_{\le n}$ but not in $b(F_{\le n})$. This illustrates a detail of the construction of $b(F)$, which is essential for the preservation of determinism (Proposition 14). This difference is inherited to accessibility in $A \times F$ and $A \times b(F)$. In order to avoid ambiguities, we write $A, F \models \mathsf{f.acc}(p,q)$ if (p,q) is accessible in $A \times F$. Thus, the following implication holds but not its converse:

$$A, b(F) \models \mathsf{acc}(p,q) \Rightarrow A, F \models \mathsf{f.acc}(p,q)$$

We define a ground Datalog program $D_0(A, F)$ in Fig. 12 in order to compute all valid acc and f.acc literals, i.e., all literals with $A, b(F) \models \mathsf{acc}(p,q)$ and $A, F \models \mathsf{f.acc}(p,q)$. Furthermore, program $D_0(A, F)$ provides rules ($\mathrm{fail}_{0/a}$) and ($\mathrm{fail}_{2/a}$), which infer literals $\mathtt{fail}_0$ and $\mathtt{fail}_2$ respectively, according to Lemma 15.

It remains to verify that the program $D_0(A, F)$ does indeed infer all valid acc and f.acc literals. This is shown by the following Lemma.

Lemma 16. *Let $B = b(F)$ and $L = lfp(D_0(A, F))$.*

1. $A, B \models \mathsf{acc}(p,q)$ *iff* $\mathsf{acc}(p,q) \in L$.
2. $A, F \models \mathsf{f.acc}(p,q)$ *iff* $\mathsf{f.acc}(p,q) \in L$.

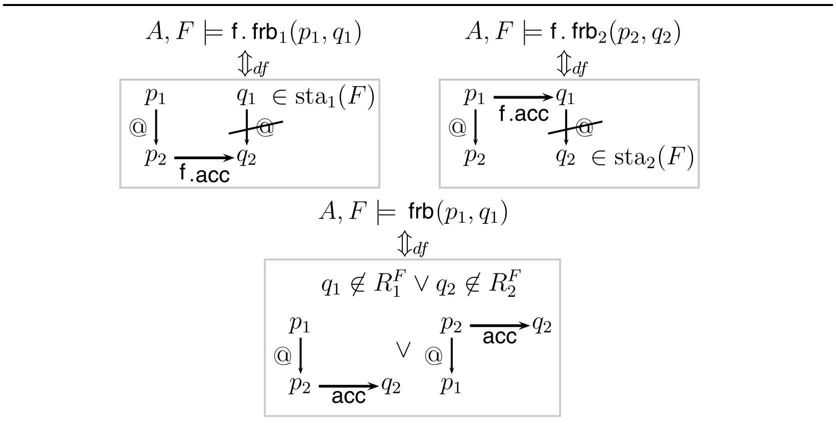


Fig. 13. Semantics of predicates $\mathsf{f.frb}_1$, $\mathsf{f.frb}_2$ and frb for factorized tree automata.

3. $A, B \models \mathtt{fail}_0$ *iff* $\mathtt{fail}_0 \in L$.
4. $A, B \models \mathtt{fail}_2$ *iff* $\mathtt{fail}_2 \in L$.

Proof. The four implications from the right to the left can be shown by simultaneous induction of the definition of the least fixed point. This is technical but straightforward. For the implications from the left to the right, we proceed as follows.

1. We show that $(p, q) \in eval_{A\times B}(t)$ implies $\mathsf{acc}(p, q) \in L$ by induction on the structure of t. Here we need construction rule (E_2) of $b(F)$.
2. We show that $(p, q) \in eval_{A\times F}(t)$ implies $\mathsf{f.acc}(p, q) \in L$ by induction on the structure of t.
3. From Lemma 15 and transformation rule $(\mathrm{fail}_{0/a})$.
4. From Lemma 15, transformation rule $(\mathrm{fail}_{2/a})$, and part (1) above. $\square$

Lemma 17. *$D_0(A, F)$ can be computed in time $O(|A| \cdot |F|)$ from A and F.*

The proof is obvious. One can even compute the least fixed point of $D_0(A, F)$ such that only the productive part of $A \times F$ has to be inspected.

Proposition 18. *We can test in time $O(|A| \cdot |F|)$ whether $A, b(F) \models \mathtt{fail}_0$ and $A, b(F) \models \mathtt{fail}_2$.*

Proof. By Lemma 17 it is sufficient to compute the least fixed point of $D_0(A, F)$ and to verify whether it contains $\mathtt{fail}_0$, resp. $\mathtt{fail}_2$. This can be done in time $O(|A| \cdot |F|)$ by Lemma 17, even such that only the productive part of $A \times F$ is inspected. $\square$

4.3. Testing validity of $\mathtt{fail}_1$

It remains to characterize $A, b(F) \models \mathtt{fail}_1$ in terms of A and F. Our solution in Lemma 19 will be technically intricate.

We need literals for A, F which are $\mathsf{f.frb}_1(p, q)$, $\mathsf{f.frb}_2(p, q)$, and $\mathsf{frb}(p, q)$, whose semantics is summarized in Fig. 13. We start with $\mathsf{f.frb}_1(p, q)$ literals:

$$A, F \models \mathsf{f.frb}_1(p_1, q_1) \Leftrightarrow_{df} \begin{cases} q_1 \in sta_1(F) \wedge \exists p_2, q_2. \\ A, F \models \mathsf{f.acc}(p_2, q_2) \wedge A \models p_1 @ p_2 \wedge F \not\models q_1 @ q_2 \end{cases}$$

Note that $A, F \models \mathsf{f.frb}_1(p, q)$ does not always imply $A, b(F) \models \mathsf{frb}(p, q)$, since we do not require $q_2 \in sta_2(F)$ in the above definition. The definition of $A, F \models \mathsf{f.frb}_2(p, q)$ is symmetric. The third predicate has the following meaning, where $R_i^F = \{q \mid \exists r \in sta_i(F).\, q \xrightarrow[F]{\epsilon}{}^{\leq 1} r\}$ for $i \in \{1, 2\}$.

$$A, F \models \mathsf{frb}(p_1, q_1) \Leftrightarrow_{df} \begin{cases} \exists q_2.\, (q_1 \notin R_1^F \vee q_2 \notin R_2^F) \wedge \exists p_2. \\ A \models (p_1 @ p_2 \vee p_2 @ p_1) \wedge A, F \models \mathsf{acc}(p_2, q_2) \end{cases}$$

As the proof of Lemma 19 will show, it holds that $A, F \models \mathsf{frb}(p, q)$ implies $A, b(F) \models \mathsf{frb}(p, q)$, but not *vice versa*.

$$(\mathsf{acc}_{/4})\ \frac{p \in \mathrm{sta}(A) \qquad q \in \mathrm{sta}(F)}{\mathsf{acc}(p, \_) :-\ \mathsf{acc}(p, q).}$$

$$(\mathsf{frb}_{/1a})\ \frac{A \models p_1 \underline{@} p_2 \qquad q_1 \notin R_1^F}{\mathsf{frb}(p_1, q_1) :-\ \mathsf{acc}(p_2, \_).}$$

$$(\mathsf{frb}_{/2a})\ \frac{A \models p_1 \underline{@} p_2 \qquad q_2 \notin R_2^F}{\mathsf{frb}(p_2, q_2) :-\ \mathsf{acc}(p_1, \_).}$$

$$(\mathrm{fail}_{1/a})\ \frac{p \in \mathrm{sta}(A) \qquad q \in \mathrm{sta}(F)}{\mathrm{fail}_1 :-\ \mathsf{frb}(p, q), \mathsf{acc}(p, q).}$$

Fig. 14. $D_1(A, F)$ extends $D_0(A, F)$ for checking `fail`$_1$ raised by $A, F \models \mathsf{frb}(p, q)$.

Lemma 19. *$A, b(F) \models \mathsf{frb}(p, q)$ iff one of following properties holds*:

1. *there exists $i \in \{1, 2\}$ such that $A, F \models \mathsf{f.frb}_i(p, r)$, where r is the unique state of sort i with $q \xrightarrow[F]{\epsilon}{}^{\leq 1} r$, or*
2. *$A, F \models \mathsf{frb}(p, q)$.*

Proof. Let $B = b(F)$.

For the implication from the left to the right, we assume $A, B \models \mathsf{frb}(p_1, q_1)$. By definition, there is a literal satisfying $A, B \models \mathsf{acc}(p_2, q_2)$ such that (a) $A \models p_1 \underline{@} p_2$ and $B \not\models q_1 \underline{@} q_2$, or (b) $A \models p_2 \underline{@} p_1$ and $B \not\models q_2 \underline{@} q_1$. By symmetry, it is sufficient to consider case (a). Part (1) of Lemma 16 shows that $A, B \models \mathsf{acc}(p_2, q_2)$ implies $A, F \models \mathsf{acc}(p_2, q_2)$. We distinguish two exhaustive cases:

1. Case $q_1 \in R_1^F \wedge q_2 \in R_2^F$. There exists a unique state $r_1 \in sta_1(F)$, resp. $r_2 \in sta_2(F)$, such that $q_1 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_1$, resp. $q_2 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_2$. In this situation, $B \not\models q_1 \underline{@} q_2$ is equivalent to $F \not\models r_1 \underline{@} r_2$. From $A, B \models \mathsf{acc}(p_2, q_2)$, it follows that $A, F \models \mathsf{f.acc}(p_2, r_2)$ and hence, $A, F \models \mathsf{f.frb}_1(p_1, r_1)$.
2. Case $q_1 \notin R_1^F \vee q_2 \notin R_2^F$. By definition, this implies $A, F \models \mathsf{frb}(p_1, q_1)$.

For the other direction, we have to consider the two cases.

1. By symmetry, we can assume $i = 1$. We thus assume that the unique $r_1 \in sta_1(F)$ with $q_1 \xrightarrow[F]{\epsilon}{}^{\leq 1} r_1$ satisfies $A, F \models \mathsf{f.frb}_1(p_1, r_1)$. By definition, there exist p_2 and r_2 such that $A \models p_1 \underline{@} p_2$ and $A, F \models \mathsf{f.acc}(p_2, r_2)$. By parts (2) and (1) of Lemma 16 there exists q_2 such that $A, B \models \mathsf{acc}(p_2, q_2)$ and $q_2 \xrightarrow{\epsilon}{}^{\leq 1} r_2$. In this situation, $F \not\models r_1 \underline{@} r_2$ is equivalent to $B \not\models q_1 \underline{@} q_2$. Hence, $A, F \models \mathsf{frb}(p_1, q_1)$.
2. We assume $A, F \models \mathsf{frb}(p_1, q_1)$ and show that $A, B \models \mathsf{frb}(p_1, q_1)$. By definition, there exist q_2 such that $q_1 \notin R_1^F \vee q_2 \notin R_2^F$ and p_2 such that $A \models p_1 \underline{@} p_2 \vee p_2 \underline{@} p_2$ and $A, B \models \mathsf{acc}(p_2, q_2)$. By symmetry, we can assume that $A \models p_1 \underline{@} p_2$. From $q_1 \notin R_1^F \vee q_2 \notin R_2^F$, it follows that $B \not\models q_1 \underline{@} q_2$ and hence, $A, B \models \mathsf{frb}(p_1, q_1)$. $\square$

Our next goal is to test $A, b(F) \models$ `fail`$_1$, when raised by $A, F \models \mathsf{frb}(p, q)$ and $A, b(F) \models \mathsf{acc}(p, q)$, in time $O(|A| \cdot |F|)$. A naive Datalog program of size $O(|A| \cdot |sta(F)|^2)$ is easy to deduce from the definition of $A, F \models \mathsf{frb}(p, q)$. The less naive Datalog program $D_1(A, F)$ in Fig. 14 extends $D_0(A, F)$, in order to solve this task in time $O(|A| \cdot |F|)$. In order to avoid the quadratic factor, it relies on new literals $\mathsf{acc}(p, \_)$, which we define to be equivalent to $\exists q.\ \mathsf{acc}(p, q)$:

$$A, b(F) \models \mathsf{acc}(p, \_) \Leftrightarrow_{df} A, F \models \exists q.\ \mathsf{acc}(p, q)$$

All valid literals of type $\mathsf{acc}(p, \_)$ are computed by $D_1(A, F)$ by clauses from $(\mathrm{acc}_{/4})$ and $D_0(A, F)$. The remaining clauses from $D_1(A, F)$ check whether `fail`$_1$ is raised by valid frb literals.

Lemma 20. *$A, F \models \exists p \exists q.\ \mathsf{frb}(p, q) \wedge \mathsf{acc}(p, q)$ iff `fail`$_1 \in lfp(D_1(A, F))$.*

Proof. The soundness ("⇐") of the rules is obvious. It remains to show their completeness ("⇒"). We assume $A, F \models \mathsf{frb}(p_1, q_1) \wedge \mathsf{acc}(p_1, q_1)$. By definition of $A, F \models \mathsf{frb}(p_1, q_1)$, this holds in situations where $A \models p_1 \underline{@} p_2$, $A, F \models \mathsf{acc}(p_2, q_2)$ and $q_1 \notin R_F^1 \vee q_2 \notin R_F^2$. For symmetry, it is sufficient to consider the case $q_1 \notin R_F^1$. Let $L = lfp(D_1(A, F))$. Part (1) of Lemma 16 shows $\mathsf{acc}(p_1, q_1) \in L$ and hence $\mathsf{acc}(p_1, \_) \in L$ by $(\mathrm{acc}_{/4})$. From this, it can be deduced $\mathsf{frb}(p_2, q_2) \in L$ by $(\mathrm{frb}_{/1a})$, so that `fail`$_1 \in L$ by $(\mathrm{fail}_{1/a})$. Note that $\mathsf{frb}(p_1, q_1) \in L$ is possible, even though $A, F \models \mathsf{frb}(p_1, q_1)$. $\square$

Lemma 21. *$D_1(A, F)$ can be computed in time $O(|A| \cdot |F|)$ from A and F.*

Proof. All clauses depend on an element of A and an element F only. $\square$

$$(\mathsf{f.frb}_1^c)\ \frac{A \models p_1 \underline{@} p_2 \quad q_2 \in \mathrm{sta}_2(F)}{\mathsf{f.frb}_1^c(p_1, Q_1^F(q_2)) :- \mathsf{f.acc}(p_2, q_2).}$$
$$(\mathsf{f.frb}_2^c)\ \frac{A \models p_1 \underline{@} p_2 \quad q_1 \in \mathrm{sta}_1(F)}{\mathsf{f.frb}_2^c(p_2, Q_2^F(q_1)) :- \mathsf{f.acc}(p_1, q_1).}$$

Fig. 15. $D_2(A, F)$ extends $D_1(A, F)$ with clauses for $\mathsf{f.frb}_i^c$.

Our next objective is to test $A, F \models \mathsf{f.frb}_i(p, q)$ for all p, q in time $O(|A| \cdot |F|)$. We consider $i = 1$ only, for the sake of symmetry. Analogically to the case without factorization, we define the following counters:

$$l_1(p) = \#\{(p', q') \mid A \models p' \underline{@} p \wedge A, F \models \mathsf{f.acc}(p', q')\}$$
$$l_1(p, q) = \#\{(p', q') \mid A \models p' \underline{@} p \wedge A, F \models \mathsf{f.acc}(p', q') \wedge F \models q' \underline{@} q\}$$

Lemma 22. *For all $q \in sta_1(F)$, $A, F \models \mathsf{f.frb}_1(p, q)$ iff $l_1(p) > l_1(p, q)$.*

Proof. By definition, $l_1(p) \geq l_1(p, q)$ for all p, q. We have $l_1(p) > l_1(p, q)$ iff $\exists p', q'$ such that $A \models p' \underline{@} p \wedge A, F \models \mathsf{f.acc}(p', q')$ $\wedge\ F \not\models q' \underline{@} q$. Since $q \in sta_1(F)$ by assumption, this is equivalent to $A, F \models \mathsf{f.frb}_1(p, q)$. $\square$

Theorem 23. *For stepwise tree automata with ϵ-rules A and deterministic factorized tree automata F over the same signature, inclusion $L(A) \subseteq L(F)$ can be decided in time $O(|A| \cdot |F|)$.*

Proof. The algorithm first computes $lfp(D_1(A, F))$ in time $O(|A| \cdot |F|)$. It returns false if the fixed point contains $\mathtt{fail}_0$, $\mathtt{fail}_1$, or $\mathtt{fail}_2$. Otherwise, it computes the values of all counters $l_i(p)$ and $l_i(p, q)$. If $l_i(p) > l_i(p, q)$ for some $\mathsf{acc}(p, q) \in lfp(D_1(A, F))$ then the algorithm returns false, otherwise true. All these steps can be performed in time $O(|A| \cdot |F|)$ as argued above. $\square$

4.4. Efficient algorithm

We present a more efficient method to compute the values of the counters, that is similar to the non-factorized case. We use new predicates $\mathsf{f.frb}_i^c$ which account for complementation with respect to sort i, where $i \in \{1, 2\}$:

$$A, F \models \mathsf{f.frb}_i^c(p, Q) \quad \Leftrightarrow_{df} \quad \forall q \in sta_i(F) \backslash Q,\ A, F \models \mathsf{f.frb}_i(p, q)$$

Datalog program $D_2(A, F)$ in Fig. 15 infers $\mathsf{f.frb}_i^c(p, Q)$ literals. It extends $D_1(A, F)$ by clauses from two further transformation rules ($\mathsf{f.frb}_i^c$).

Lemma 24. *If $D = D_2(A, F)$ then $l_i(p) = \sum_{Q \subseteq sta_i(F)} lfp^{\#}(D)(\mathsf{f.frb}_i^c(p, Q))$ and $l_i(p, q) = \sum_{Q \subseteq sta_i(F), q \in Q} lfp^{\#}(D)(\mathsf{f.frb}_i^c(p, Q))$.*

Proof. Similar to the proof of Lemma 12. $\square$

Lemma 25. *$D_2(A, F)$ can be computed in time $O(|A| \cdot |F|)$ from A and F.*

Proof. The proof works as for $D_2(A, B)$ in Lemma 10. The grouping clauses produced by ($\mathsf{f.frb}_i^c$) can be rewritten in analogy to those for (frb^c) before. The sets R_i^F are of size $O(|B|)$ and occur at most $O(|A|)$ times in rules (frb_i^c), thus the overall size of the clauses produced by this rule is in $O(|A| \cdot |B|)$, too. The analysis for the remaining rules is straightforward. $\square$

An example for the algorithm is given in Fig. 16. Automaton A given there recognizes all trees, and the factorized tree automata $F_{\leq n}$ all Curried encodings of unranked trees (see Section 5 for the definitions) with at most n children per node. Datalog program $D_2(A, F_{\leq n})$ infers the literals $\mathsf{f.frb}_1^c(\mathrm{all}, \{0, \ldots, n{-}1\})$ and $\mathsf{f.frb}_2^c(\mathrm{all}, \{\mathrm{ok}\})$ as illustrated on the right of the figure. The first implies $A, b(F_{\leq n}) \models \mathsf{f.frb}_1(\mathrm{all}, n)$ and thus $A, b(F_{\leq n}) \models \mathtt{fail}_1$. The second is a tautology since $sta_2(F_{\leq n}) = \{\mathrm{ok}\}$.

Compared to the non-factorized case, our algorithm cannot infer $\mathsf{frb}^c(p, Q)$ literals efficiently any more. This would require to apply epsilon rules over and over, spoiling our time complexity of $O(|A| \cdot |F|)$. Instead, our algorithm infers $\mathsf{f.frb}_i^c(p, Q)$ literals and combines them with $\mathsf{f.acc}(p, q)$ literals from ($\mathsf{f.frb}_i^c$) in Fig. 15. Epsilon rules are used for inferring the $\mathsf{f.acc}(p, q)$ literals (see Fig. 12). Besides, they only serve in transformation rules ($\mathsf{frb}_{/ia}$) from Fig. 14, which deal with cases where evaluation stops in some state that cannot be converted to the required sort by any ϵ-rule.

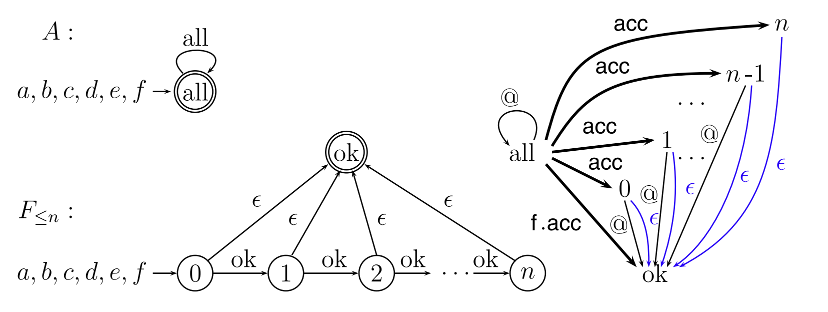


Fig. 16. Example run of the algorithm: $\mathsf{f.frb}_1^c(\mathrm{all}, \{0, \ldots, n-1\})$ is inferred.

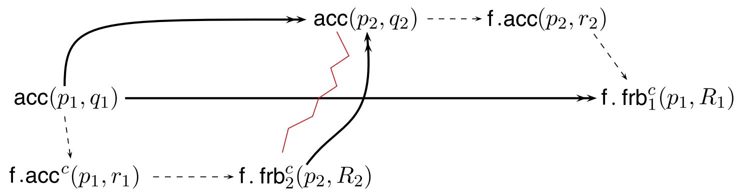


Fig. 17. Early failure detection for $i = 1$.

4.5. Early failure detection

We show how to check for $\mathtt{fail}_1$ on the fly. We assume that all counters $l_i(p)$ and $l_i(p, q)$ are always up to date. As in the non-factorized case, we assume that literals with predicates acc are inferred with the lowest priority (also lower than $\mathsf{f.acc}$).

The *on-the-fly algorithm* works as follows. It computes $lfp(D_2(A, F))$ while returning false once $\mathtt{fail}_0$, $\mathtt{fail}_1$ or $\mathtt{fail}_2$ is produced. The counters $l_i(p)$ and $l_i(p, q)$ are always kept up-to-date, i.e., increased when some literal $\mathsf{frb}^c(p, Q)$ is inferred by some further clause. For all newly inferred literals $\mathsf{acc}(p, q)$, it checks whether some literal $\mathsf{f.frb}_i(p, Q)$ was produced before, where $i \in \{1, 2\}$ and $r \in sta_i(F) \backslash Q$ with $q \overset{\epsilon}{\to}^{\leq 1} r$. The existence of a literal $\mathsf{f.frb}_i(p, Q)$ is reduced to checking whether $l_1(p) > l_1(p, r)$ or $l_2(p) > l_2(p, r)$. If so, $\mathtt{fail}_1$ is raised and the algorithm returns false. Otherwise, it continues with the fixed point computation. Testing the counters on the fly is sufficient, as shown by the following lemma.

Lemma 26. *The on-the-fly algorithm detects* $\mathtt{fail}_1$ *if* $\mathsf{acc}(p, q)$ *is inferred before some literal* $\mathsf{f.frb}_i^c(p, Q)$, *where* $r \notin Q$ *is the unique state with* $q \overset{\epsilon}{\to}^{\leq 1} r$.

Proof. We consider the case $i = 1$ only, which is sufficient by symmetry. Let $\mathsf{f.frb}_1^c(p_1, R_1)$ be inferred after $\mathsf{acc}(p_1, q_1)$, where $q_1 \overset{\epsilon}{\to}_F^{\leq 1} r_1$ and $r_1 \notin sta_1(F) \backslash R_1$. See Fig. 17 for illustration. Literal $\mathsf{f.frb}_1^c(p_1, R_1)$ is justified by some literal $\mathsf{f.acc}(p_2, r_2)$ added before, and the clause below where $R_1 = Q_1^F(r_2)$. Furthermore, $\mathsf{f.acc}(p_2, r_2)$ stems from some literal $\mathsf{acc}(p_2, q_2)$ and the second clause:

$$\frac{A \models p_1@p_2 \qquad r_2 \in sta_2(F)}{\mathsf{f.frb}_1^c(p_1, R_1) :- \mathsf{f.acc}(p_2, r_2).} \qquad \frac{p_2 \in sta(A) \qquad q_2 \overset{\epsilon}{\to}^{\leq 1} r_2}{\mathsf{f.acc}(p_2, r_2) :- \mathsf{acc}(p_2, q_2).}$$

Due to the lowest priority of acc literals again, $\mathsf{acc}(p_1, q_1)$ must be inferred before $\mathsf{acc}(p_2, q_2)$. The following clauses can be applied where $R_2 = Q_2^F(r_1)$:

$$\frac{p_1 \in sta(A) \qquad q_1 \overset{\epsilon}{\to}^{\leq 1} r_1}{\mathsf{f.acc}(p_1, r_1) :- \mathsf{acc}(p_1, q_1).} \qquad \frac{A \models p_1@p_2 \qquad r_1 \in sta_1(F)}{\mathsf{f.frb}_2^c(p_2, R_2) :- \mathsf{f.acc}(p_1, r_1).}$$

Since $r_1 \in Q_1^F(r_2)$ if and only if $r_2 \in Q_2^F(r_1)$, it follows from $r_1 \notin R_1$ that $r_2 \notin R_2$. Thus, $\mathsf{acc}(p_2, q_2)$ and $\mathsf{f.frb}_2^c(p_2, R_2)$ in $lfp(D_2(A, F))$ raise inclusion failure $\mathtt{fail}_1$. By priority, $\mathsf{f.frb}_2^c(p_2, R_2)$ is added before $\mathsf{acc}(p_2, q_2)$, so this failure is properly detected by the incremental algorithm. $\square$

As in the non-factorized case, we can turn this algorithm incremental with respect to adding epsilon edges to A. We can thus test inclusion in deterministic factorized automata as efficiently as for the non-factorized case.

5. DTDs and factorized tree automata for unranked trees

We lift our inclusion test to factorized tree automata interpreted over unranked trees, so that it becomes applicable to deterministic DTDs. Factorization is essential for efficiency here.

5.1. Factorized tree automata for unranked trees

An unranked signature Σ is a finite set of symbols (without arity restrictions). The set T^u_Σ of unranked trees over Σ is the least set containing all tuples $a(t_1,\ldots,t_n)$ where $a \in \Sigma, t_1,\ldots,t_n \in T^u_\Sigma$ and $n \geq 0$.

Currying carries over literally from ranked to unranked trees. This yields the bijective function $curry : T^u_\Sigma \to T_{\Sigma_@}$, which satisfies $curry(a(t_1,\ldots,t_n)) = a@curry(t_1)@\ldots@curry(t_n)$ for all unranked trees $a(t_1,\ldots,t_n) \in T^u_\Sigma$. For instance, $curry(a(b,c,d(e))) = a@b@c@(d@e)$. Subtrees of $a(b,c,d(e))$ are encoded as subtrees on the right of @ such as $d@e$. Subtrees on the left of @ encode rooted hedges such as $a@b@c$ that are subject to extension to the right. This semantic difference motivated different sorts for hedges and unranked trees already in the automata notions of Raeymaekers [14] or Neumann and Seidl [4].

We can use factorized tree automata A over stepwise signatures Σ to recognize languages of unranked trees $L^u(A) = \{t \in T^u(\Sigma) \mid curry(t) \in L(A)\}$. We obtain the following corollary from Theorem 23.

Corollary 27. *Let A be a stepwise tree automaton and F a deterministic factorized tree automaton over the same signature Σ. Language inclusion $L^u(A) \subseteq L^u(F)$ can be decided in time $O(|A| \cdot |F|)$ independently of $|\Sigma|$.*

The tree automaton A can also be chosen to be a hedge automaton, whose horizontal languages are defined by non-deterministic finite word automata (nFAs). Hedge automata H over Σ have rules of the form $a(C) \to q$ where $a \in \Sigma$, $q \in sta(H)$, and C is an nFA with signature $sta(H)$. Such hedge automata are called NFHAs by Comon et al. [1] and UTAs by Martens and Niehren [13]. They can be translated in linear time to stepwise tree automata with ϵ-rules [13]. A hedge automaton is called deterministic if all its nFAs are deterministic (dFAs) and $L(q_1) \cap L(q_2) = \emptyset$ for all two rules $a(C_1) \to q_1$ and $a(C_2) \to q_2$ in $rul(H)$. This is only a pseudo-notion of determinism. It is mapped to unambiguity of stepwise tree automata. As a consequence, we cannot choose F to be a deterministic hedge automaton.

5.2. Deterministic DTDs

We convert deterministic DTDs D to deterministic factorized tree automata for unranked trees in time $O(|\Sigma| \cdot |D|)$, so that we can reuse our algorithm for testing inclusion of stepwise tree automata in deterministic DTDs. Here, factorization avoids the quadratic blowup. When translating into stepwise tree automata [13], the number of rules may become quadratic, while the number of states is preserved. The problem is the implicit elimination of ϵ-rules.

A DTD D with elements in a set Σ is a function mapping letters $a \in \Sigma$ to regular expressions e over Σ, in which case we write $a \to_D e$. One of these elements is the distinguished start symbol. Let $L(e) \subseteq \Sigma^*$ be the word language defined by e. The language $L_a(D) \subseteq T^u_\Sigma$ of elements a of a DTD D is the smallest set of unranked trees such that:

$$L_a(D) = \{a(t_1,\ldots,t_n) \mid a \to_D e, \quad a_1\ldots a_n \in L(e), \quad t_i \in L_{a_i}(D) \text{ for } 1 \leq i \leq n\}$$

The language of a DTD D is $L(D) = L_a(D)$ where a is the start symbol of D. The size of D is the total number of symbols in the regular expressions of D. An example with its corresponding XML syntax is given in Fig. 18. The set of elements of D is $\Sigma = \{\texttt{doc}, \texttt{block}, \texttt{text}, \texttt{link}\}$, of which the element `doc` is the start symbol. The regular expression for `#PCDATA` recognizes only the empty word.

A DTD is deterministic if all its regular expressions are one-unambiguous, as required by the W3C. This is equivalent to say that all corresponding Glushkov automata are deterministic [12]. Glushkov automata are nFAs from regular expressions as usual except that ϵ rules are eliminated on the fly whenever they appear. The precise definition is outside the scope of this article, but an example is given in Fig. 18.

Theorem 28 (Brüggemann-Klein [28]). *The collection of Glushkov automata for a deterministic DTD D over Σ can be computed in time $O(|\Sigma| \cdot |D|)$.*

Note that the construction of the Glushkov automaton of a regular expression e over alphabet Σ may take time $O(|\Sigma| \cdot |e|^2)$ in the general case. Intuitively, the square factor is raised by eliminating occurring ϵ-rules on the fly. In the case of a one-unambiguous regular expression, the resulting Glushkov automaton is deterministic. The construction time is bounded by its size and thus in $O(|\Sigma| \cdot |e|)$ due to determinism.

```
doc    →  block+
block  →  text (link text?)?
          + link text?
text   →  ϵ
link   →  ϵ
```

```
<!ELEMENT doc  (block+)>
<!ELEMENT block (text ,( link , text ?)?
                 | link , text ?)>
<!ELEMENT text  (#PCDATA)>
<!ELEMENT link  (#PCDATA)>
```

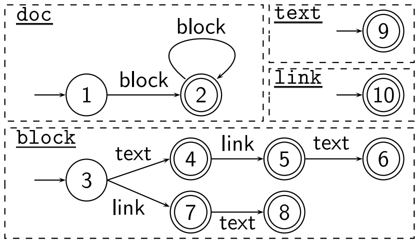


Fig. 18. An example DTD and the corresponding Glushkov automata.

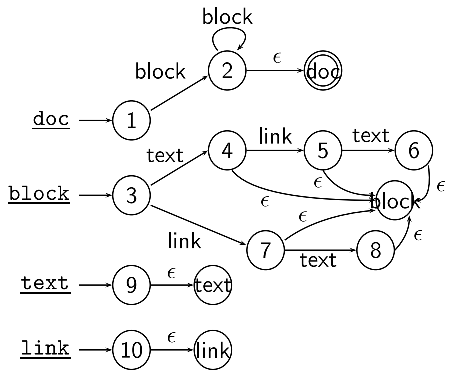


Fig. 19. A representation of the deterministic factorized tree automaton for the DTD in Fig. 18. Alphabet Σ is {doc,block,text,link}, states of sort 1 are $\{1, \ldots, 10\}$ and states of sort 2 are {doc,block,text,link}. A constant rule is for instance doc $\to$ 1, a binary rule 2@block $\to$ 2 and an ϵ-rule $2 \xrightarrow{\epsilon}$ doc.

We transform the collection of Glushkov automata for a deterministic DTD D into a single factorized tree automaton F as follows. The set of states of sort 1 of F is the disjoint union of the states of the Glushkov automata. The states of sort 2 of F are the elements of D. For every element a, we connect all final states q of its Glushkov automaton to the state a, i.e., $q \xrightarrow{\epsilon} a \in rul(F)$. The only final state of F is the start symbol of the DTD D. The result is an nFA that represents a factorized tree automaton, as for instance in Fig. 19. This needs time of at most $O(|\Sigma| \cdot |D|)$. For every $a \in \Sigma$, there is a rule $a \to q \in rul(F)$ for the unique initial state q of the Glushkov automaton of a. For every transition $q \xrightarrow{a} q'$ of one of the Glushkov automata, we add a rule $q@a \to q' \in rul(F)$.

Note that F is deterministic as a factorized automaton. The ϵ-free part of F is deterministic since all Glushkov automata are, thus establishing $\mathtt{d}_0$. Let q be a state of the Glushkov automaton for some letter a. The only state of sort 1 that q can reach by ϵ-edges in F is a and the only state of sort 2 is q itself. All other states of F are elements of $a \in \Sigma$, which have no outgoing ϵ-edges, thus establishing $\mathtt{d}_1$. Note that the size of the example automaton would grow quadratically, when eliminating ϵ-edges.

Theorem 29. *Deterministic DTDs D over Σ can be translated in time $O(|\Sigma| \cdot |D|)$ to bottom-up deterministic factorized tree automata recognizing the same language.*

Proof. The translation of a collection of Glushkov automata of a DTD to a factorized automaton is in linear time. It is easy to check that it preserves the languages of unranked trees. The theorem thus follows from Theorem 28 by Brüggemann-Klein. □

Corollary 30. *Language inclusion of hedge automata A over Σ with horizontal languages defined by finite word automata in deterministic DTDs D with elements in Σ can be decided in time $O(|A| \cdot |\Sigma| \cdot |D|)$.*

Proof. From Corollary 27 and Theorem 29. □

6. Experiments

We have implemented the inclusion algorithm in Objective CAML, and have integrated it into a system for schema-guided learning of queries in XML trees [9]. In the first set of experiments, we consider inclusion tests for synthetic automata. Then we consider inclusion tests between automata and DTDs coming from realistic tasks in query learning.

6.1. Experiment 1

We modify the sizes of automata A and F when testing inclusion of $L(A)$ in $L(F)$. For this, we define $Mult_n$ as the minimal deterministic automaton for the language of trees of the form $f(a, \ldots, a)$ where the number of a-leaves is a multiple of n. The first problem is to test inclusion of $L(Mult_n)$, n varying from 100 to 10,000 with a 100-increment, into the minimal deterministic factorized automaton recognizing $L(Mult_{200})$. It should be noted that inclusion holds when $n/100$ is even. The

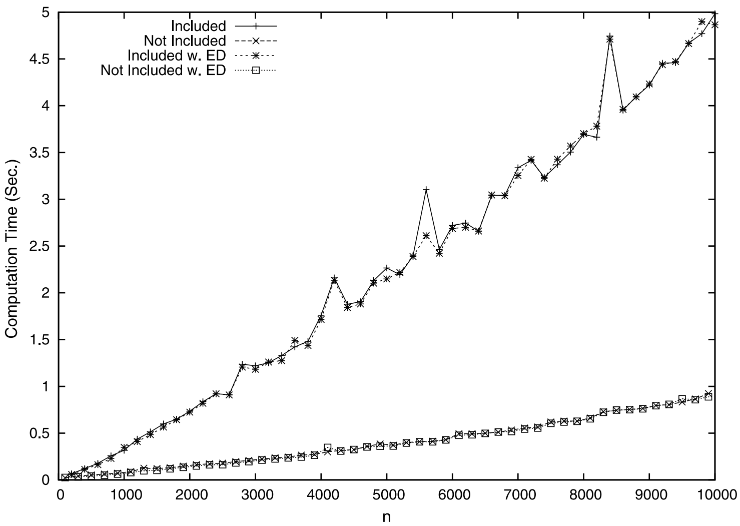


Fig. 20. Computation time for testing $L(Mult_n) \subseteq L(Mult_{200})$.

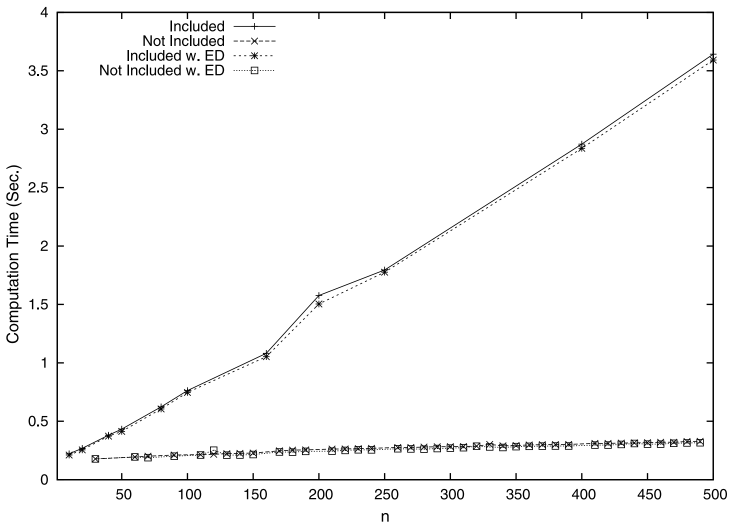


Fig. 21. Computation time for testing $L(Mult_{400}) \subseteq L(Mult_n)$.

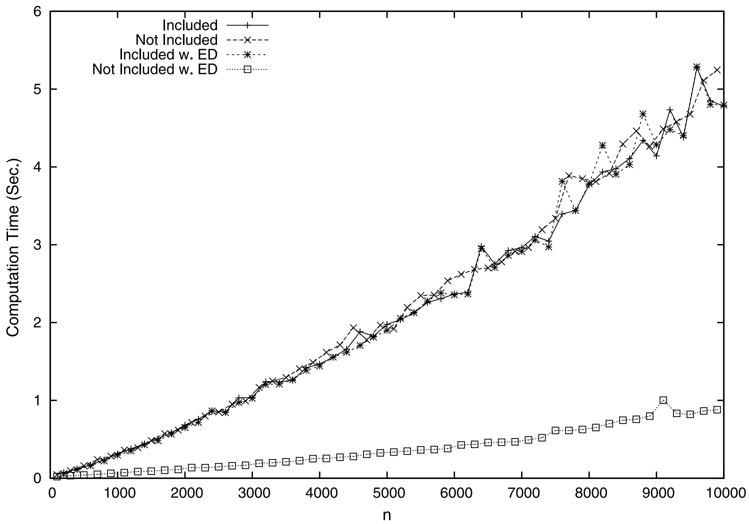


Fig. 22. Average computation time for testing $L(Mult2_n) \subseteq L(Mult2_{200})$.

second problem is to test inclusion of $L(Mult_{400})$ into $L(Mult_n)$ with n varying from 10 to 500 with a 10-increment. It should be noted that inclusion holds when $400/n$ is an integer.

We estimate the computation time for inclusion tests with and without early detection of inclusion failure (ED). We distinguish whether inclusion holds or not. Results are shown in Fig. 20 for the first problem and in Fig. 21 for the second problem.

It can be verified that the computation time of testing $L(A) \subseteq L(F)$ is linear in the size of the automaton A and in the size of automaton F. This confirms the theoretical results on complexity. It can also be seen that the computation time is greater when inclusion holds. Otherwise, the computation time is lower since concurrent failure detection applies. In this experiment, there are no failures of type $\mathtt{fail}_1$, so we do not use early failure detection. The gain is obtained by checking $\mathtt{fail}_2$ concurrently, so that product automaton does not need to be computed entirely.

6.2. Experiment 2

In order to verify the usefulness of early detection of failures of type $\mathtt{fail}_1$ (ED), we consider another example. We define $Mult2_n$ to be the minimal deterministic automaton for the language of trees of the form $g(f(a, \ldots, a))$, where the number of a-leaves is a multiple of n. The problem is to test the inclusion of $L(Mult2_n)$, where n varies from 100 to 10,000 with a 100-increment, into the minimal deterministic factorized automaton recognizing $L(Mult2_{200})$. The computation times are shown in Fig. 22.

It can be noted that when inclusion does not hold, computation time is five times faster than for other cases. This is because, for these inclusion tests, inclusion failure comes from $\mathtt{fail}_1$. Thus early failure detection allows to decrease dramatically the computation time. It can also be noted that the computation time is similar for cases where inclusion is verified (with or without early detection), and non-inclusion cases without early detection. This is because, without early failure detection, computing a failure $\mathtt{fail}_1$ implies the need to compute the whole product automaton.

6.3. Experiment 3

We now consider real-world data sets from the query induction problem. In the learning algorithm defined by Champavère et al. [9], an initial automaton is computed and is iteratively refined by merging states. A merge is accepted only if the language recognized by the new automaton still satisfies a given schema or DTD. Consequently, inclusion tests are done frequently. We compare the overall computation time of learning sessions where inclusion tests are done with or without early failure detection. We use the the transitional DTD of XHTML and the query learning benchmarks Okra, Bigbook, Google and Yahoo, each of them with an increasing size of inputs. Results are shown in Fig. 23.

It appears that the learning algorithm operates about twice as fast with early detection of failure $\mathtt{fail}_1$ than without in all benchmarks. This indicates that $\mathtt{fail}_1$ occurs frequently in practice and that early detection improves efficiency a lot. More generally, it shows that the inclusion algorithm presented here can be used for real-world problems. In Champavère et al. [9], we have shown that introducing inclusion tests does not increase computation time while avoiding useless state merges, thus improving the query induction algorithm.

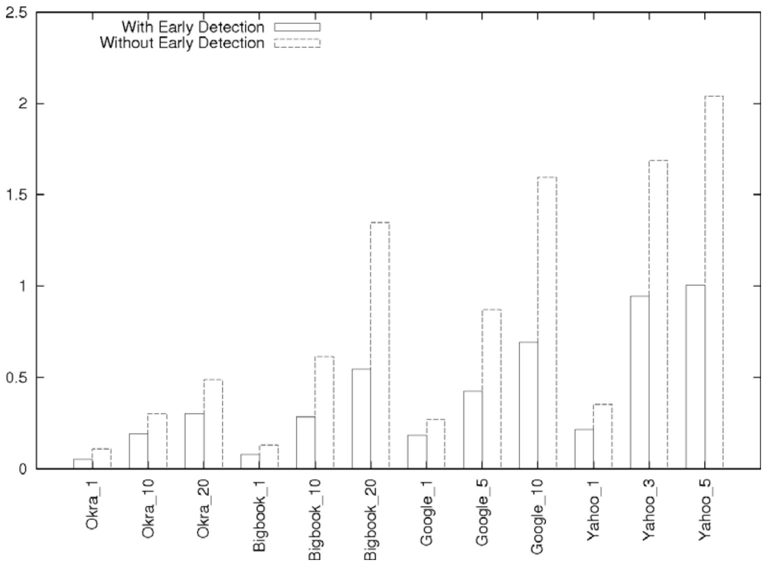


Fig. 23. Average computation time (in seconds) of learning sessions with and without early detection of failure $\texttt{fail}_1$.

7. Top-down determinism

We show how to test inclusion for top-down deterministic tree automata by reduction to inclusion testing in deterministic word automata. This reduction should be folklore (but we are not aware of any reference). For our precise complexity analysis, however, we use Theorem 9 for the case of words, seen as trees over monadic signatures.

Thereby we obtain an efficient test for inclusion in deterministic EDTDs with restrained competition (and thus for schema definitions in XML Schema), as these can be translated to top-down deterministic tree automata with respect to Rabin's firstchild-nextsibling encoding of unranked trees (see below). More precisely, we can test language inclusion for tree automata A recognizing firstchild-nextsibling encodings of unranked trees in deterministic restrained competition EDTDs D in time $O(|A| \cdot |\Sigma| \cdot |D|)$. We do not know whether the analogous result holds for automata A recognizing Curried encodings.

A restriction of essentially the same algorithm for testing inclusion between two deterministic restrained competition EDTDs was presented earlier by Martens et al. [18] (see Section 10 of the reference). Their presentation, however, does not rely on top-down deterministic tree automata as an intermediate step, and no precise complexity analysis is given.

7.1. Top-down deterministic tree automata for ranked trees

A tree automaton A over a ranked signature Σ is *top-down deterministic* if for all symbols $f \in \Sigma$ of arity n and states $p \in \Sigma$ there are no two different rules $f(p_1, \ldots, p_n) \to p$ and $f(p'_1, \ldots, p'_n) \to p$ in $rul(A)$.

Proposition 31. *Let Σ be a ranked signature, and A and B be tree automata over Σ. If B is top-down deterministic, then we can decide language inclusion $L(A) \subseteq L(B)$ in time $O(|A| \cdot |B|)$.*

We base the algorithm on the well-known fact that tree languages recognized by top-down deterministic tree automata are path-closed [30,1]. The standard example for a non-path-closed regular language is $L_0 = \{f(a,a), f(b,b)\}$ where $a \neq b$. For the sake of completeness, let us recall the definitions. The set of paths of a tree $t \in T_\Sigma$ is the subset of words $paths(t) \subseteq (\Sigma \cup \mathbb{N})^*$ defined as follows:

$$paths(a) = a, \quad \text{and} \quad paths(f(t_1, \ldots, t_n)) = \{fiw \mid 1 \leq i \leq n,\ w \in paths(t_i)\}.$$

For instance $paths(L_0) = \{f1a, f2a, f1b, f2b\}$. The path closure of a tree language $L \subseteq T_\Sigma$ is the set of all trees that contain only paths of trees in L:

$$path\text{-}clos(L) = \{t \mid paths(t) \subseteq paths(L)\}$$

We call L path-closed if $L = path\text{-}clos(L)$. For instance $path\text{-}clos(L_0) = L_0 \cup \{f(a,b), f(b,a)\}$, so L_0 is indeed not path-closed.

Lemma 32. *If L_2 is path-closed then $L_1 \subseteq L_2$ iff $paths(L_1) \subseteq paths(L_2)$.*

Proof. Note that we do not assume L_1 to be path-closed. The implication from left to right is trivial. For the inverse, assume $paths(L_1) \subseteq paths(L_2)$. If $t_1 \in L_1$ then $paths(t_1) \subseteq paths(L_1) \subseteq paths(L_2)$. Thus $t_1 \in path\text{-}clos(L_2)$, and this set is equal to L_2 by assumption of path-closeness. □

Proof of Proposition 31. For every tree automaton A over Σ, we construct an nFA $P(A)$ over a finite subset of $\Sigma \uplus \mathbb{N}$ such that $L(P(A)) = paths(L(A))$. The rules of $P(A)$ are defined as follows:

$$rul(P(A)) = \{p \xrightarrow{fi} p_i \mid f(p_1, \ldots, p_n) \to p \in rul(A),\quad 1 \le i \le n\} \cup \{a \to p \mid a \to p \in rul(A)\} \cup \{p \xrightarrow{\epsilon} p' \mid p \xrightarrow{\epsilon} p' \in rul(A)\}$$

The first kind of rules reads two letters at the same time, but can be easily rewritten into two rules reading each a single letter. Clearly, the construction of $P(A)$ is in time $O(|A|)$. Furthermore, $P(A)$ is deterministic iff A is top-down deterministic.

Given two tree automata A, B over Σ such that B is top-down deterministic, we can decide language inclusion between A and B by testing language inclusion for $P(A)$ and $P(B)$. Since $P(B)$ is deterministic this can be done in time $O(|P(A)| \cdot |P(B)|)$ independently of the alphabet by Theorem 9, which is in time $O(|A| \cdot |B|)$ independently of the signature. □

7.2. Restrained competition EDTDs

We test inclusion in restrained competition EDTDs, for tree automata recognizing unranked trees modulo the firstchild-nextsibling encoding of unranked trees. It is obtained by encoding restrained competition EDTDs to top-down deterministic tree automata with respect to this binary encoding.

An extended DTD (EDTD) D over a signature Σ consists of a finite set of states $sta(D) \subseteq \Sigma \times \mathbb{N}$, a subset of start states $start(D) \subseteq sta(D)$, and a collection of rules given by a function mapping states $q \in sta(D)$ to regular expressions e over $sta(D)$, in which case we write $q \to_D e$. The language $L_q(D) \subseteq T^u_\Sigma$ of a state $q \in sta(D)$ is the smallest set of unranked trees such that if $q = (a, i)$ for some $a \in \Sigma, i \in \mathbb{N}$ then:

$$L_q(D) = \{a(t_1, \ldots, t_n) \mid q \to_D e,\quad q_1 \ldots q_n \in L(e),\quad t_i \in L_{q_i}(D) \text{ for } 1 \le i \le n\}$$

The language of an EDTD is $L(D) = \cup_{q \in start(D)} L_q(D)$. The size of D is the total number of symbols in the regular expressions of D. Essentially, EDTDs are the same as hedge automata with regular expressions for defining horizontal languages. They can thus recognize all regular languages of unranked trees.

An EDTD D is *restrained competition* if it has a unique start state and for all regular expressions $q \to_D e$ of D there exist no two different states $(a, n_1), (a, n_2) \in sta(D)$ and words $u, v_1, v_2 \in sta(D)^*$ such that $u(a, n_1)v_1,\ u(a, n_2)v_2 \in L(e)$. Also, as for DTDs, we call a restrained competition EDTD *deterministic* if all its regular expressions are one-unambiguous and if it has at most one start state.

Restrained competition EDTDs are strictly more expressive than deterministic DTDs. On the other hand, they are more restrictive than the class of regular languages, in order to permit the typing of all nodes of an XML document in 1-pass streaming manner [31,18]. Consider for instance the regular language of unranked trees $L_1 = \{a(a)\}$. It cannot be recognized by any DTD, since it contains a tree with two types of a-nodes that need to be distinguished. Language L_1 can be recognized by a restrained competition EDTD with two states $(a, 1)$ and $(a, 2)$ for the two types of a-nodes. The start state is $(a, 1)$ and the rules are as follows:

$$(a, 1) \to (a, 2), \qquad (a, 2) \to \epsilon$$

This example illustrates that we cannot translate restrained competition EDTDs to bottom-up deterministic stepwise tree automata in linear time (in contrast to the case of deterministic DTDs). The naive approach would be to permit tree automaton rules $a \to (a, 1)$ and $a \to (a, 2)$ but these violate bottom-up determinism. The problem is that the type of an a-node is only determined once knowing the type of its parent, so we have to try out all choices in a bottom-up deterministic manner.

Let $\Sigma^\# = \Sigma \uplus \{\#\}$ be the ranked signature with a single constant # and a collection of binary function symbols $a \in \Sigma$. Rabin's firstchild-nextsibling encoding *fcns* of an unranked tree $t \in T^u(\Sigma)$ is a binary tree in $T_{\Sigma^\#}$ (see, e.g., Gott-

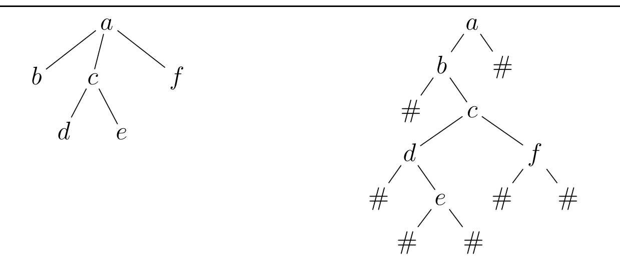


Fig. 24. The tree $a(b, c(d, e), f)$ and its firstchild-nextsibling encoding $a(b(\#, c(d(\#, e(\#, \#)), f(\#, \#))), \#)$.

lob and Koch [29]), for instance, $fcns(a(b, c(d, e), f))) = a(b(\#, c(d(\#, e(\#, \#)), f(\#, \#))), \#)$ as illustrated in Fig. 24. A tree automaton A over $\Sigma^\#$ recognizes unranked trees modulo this other binary encoding, so its unranked tree language is $L^u(A) = \{t \in T^u(\Sigma) \mid fcns(t) \in L(A)\}$.

Lemma 33. *For all deterministic restrained competition EDTDs D over Σ, we can compute a top-down deterministic tree automaton B over $\Sigma^\#$ with the same unranked tree language $L^u(B) = L(D)$ in time $O(|\Sigma| \cdot |D|)$.*

Proof. We first compute the collection of Glushkov automata G_q for all regular expressions e such that $q \to_D e$. Since D is deterministic, all regular expressions e are unambiguous, so that all Glushkov automata G_q are dFAs of overall size $O(|\Sigma| \cdot |D|)$ by Theorem 28. The alphabets of G_qs is $sta(D) \subseteq \Sigma \times \mathbb{N}$. Without restriction of generality, we can assume that G_q is productive. For productive G_q, restrained competition of D implies that there are no two rules $p \overset{(a,i)}{\to} p' \in rul(G_q)$ and $p \overset{(a,j)}{\to} p'' \in rul(G_q)$ with $i \neq j$. Determinism of G_q implies that:

(*) for all $p \in sta(G_q)$ and letters $a \in \Sigma$ there is at most one pair (i, p') such that $p \overset{(a,i)}{\to} p' \in rul(G_q)$.

From the collection $(G_q)_{q \in sta(D)}$, we build a tree automaton B over the signature $\Sigma^\#$ with $L^u(B) = L(D)$. The states of automaton B are the elements in $\{S, H\} \uplus (\uplus_{q \in sta(D)} sta(G_q))$, of which only S is final, i.e., the state of the root. State H is the state of the hash symbol # at the second child of the root. The rules in $rul(B)$ are defined as follows, where I is the unique initial state of $init(G_{(a^s,i^s)})$ and (a^s, i^s) the unique start state of D.

$$\frac{p \overset{(a,i)}{\to} p' \in rul(G_q) \qquad init(G_{(a,i)}) = \{p''\}}{a(p'', p') \to p} \qquad \frac{p \in fin(G_q)}{\# \to p} \qquad \frac{true}{\begin{array}{c} a^s(I, H) \to S \\ \# \to H \end{array}}$$

Automaton B is top-down deterministic by (*) and recognizes $L(D)$. The construction is in linear time in the size of the collection of Glushkov automata, so the overall construction requires time $O(|\Sigma| \cdot |D|)$. □

Corollary 34. *For tree automata A over $\Sigma^\#$ and deterministic restrained competition EDTDs D over Σ, language inclusion $L^u(A) \subseteq L(D)$ can be tested in time $O(|A| \cdot |\Sigma| \cdot |D|)$.*

Proof. We transform D into a top-down deterministic tree automaton B over $\Sigma^\#$ that recognizes the same unranked tree language by Lemma 33. This takes time $O(|\Sigma| \cdot |D|)$, so the size of B is in $O(|\Sigma| \cdot |D|)$ as well. We then test $L(A) \subseteq L(B)$, which can be done in time $O(|A| \cdot |\Sigma| \cdot |D|)$ by Proposition 31, since B is top-down deterministic. □

8. Conclusion

We have presented new efficient algorithms for testing language inclusion in deterministic tree automata and XML Schema definitions.

Our main contribution is an efficient inclusion test of tree automata A in bottom-up deterministic factorized tree automata B in time $O(|A| \cdot |B|)$. We have introduced early failure detection, which gives us the ability to turn our algorithm incremental with respect to adding epsilon rules to A. We have implemented our inclusion test, given experimental evidence for its efficiency, and applied it in schema-guided query induction [9]. Incrementality considerably improves efficiency according to our experiments.

We have translated deterministic DTDs D to bottom-up deterministic factorized tree automata of size $O(|\Sigma| \cdot |D|)$. Our new notion of factorization is essential for efficiency here. As a corollary, we can check inclusion of stepwise tree automata A in deterministic DTDs D in time $O(|A| \cdot |\Sigma| \cdot |D|)$. Automata A can also be chosen to be hedge automata with nFAs for defining horizontal languages [1]. Such hedge automata are equivalent to EDTDs, since regular expressions can be translated to nFAs with ϵ-rules in linear time. They can also be obtained from schema definitions in Relax NG.

We have presented a simpler inclusion test for inclusion in top-down deterministic tree automata in the case of ranked trees. This case is of interest for unranked trees, since deterministic DTDs and restrained competition EDTDs can be translated to top-down deterministic tree automata with respect to the firstchild-nextsibling encoding of unranked trees.

Future work. One question on deterministic inclusion is left open, which is whether one can test inclusion of stepwise tree automata A in restrained competition EDTDs D over the same signature Σ in time $O(|A| \cdot |\Sigma| \cdot |D|)$. There are two difficulties here. First, it does not help to convert a stepwise tree automaton into a hedge automaton or a tree automaton operating on unranked trees modulo the firstchild-nextsibling encoding, since the known transformations introduce quadratic blowups. Second, we cannot represent the collection of Glushkov automata of a deterministic restrained competition EDTD by a deterministic stepwise tree automaton of the same size. The difference is that stepwise tree automata operate bottom-up while restrained competition EDTDs work top-down. We hope to solve this problem in future work by studying inclusion in deterministic streaming tree automata [32,33,4], which can operate in a mixed top-down and bottom-up manner.

Acknowledgments

We thank Sławek Staworko for having pointed out the alternative algorithm for inclusion checking for top-down deterministic tree automata and its relevance for DTDs.

We thank the anonymous reviewers for their valuable suggestions, which helped us a lot in order to simplify the presentation of the tedious counting arguments in the core of our algorithm and proofs.

The work was supported by the ANR project Codex.

Appendix A. Implementation

We present more details on a concrete implementation of the inclusion test for factorized tree automata of Section 4. The same implementation can be used for tree automata without factorization, after conversion into factorized automata. We use a pseudo-functional programming language with imperative state, and have used Objective CAML for implementation in practice.

For simplicity, we restrict ourselves to an inclusion test without dynamic addition of new automata rules. It is not difficult, however, to make the same algorithm incremental in that respect, by returning the complete data structures at the end of the computation, rather than a Boolean value only.

The algorithm applies function `inclusion(A,F)` in Fig. 25 to a productive stepwise tree automaton A with ϵ-rules and a deterministic factorized tree automaton F. It computes the least fixed point of $D_2(A,F)$ by saturation. The two failure conditions $\mathtt{fail}_0$ and $\mathtt{fail}_2$ are covered by saturation rules ($\mathrm{fail}_{0/a}$) and ($\mathrm{fail}_{2/a}$). The other failure condition $\mathtt{fail}_1$ is tested either by saturation rule ($\mathrm{fail}_{1/a}$), or by using early detection as argued in Section 4.5. Once a failure is detected, an exception is raised in order to exit the saturation loop.

As stated in Section 4.5, $l_i(p)$ counts the number of literals $\mathsf{f.frb}_i^c(p,Q)$ inferred so far, and counter $l_i(p,q)$ the number of occurrences of $q \in Q$ in literals $\mathsf{f.frb}_i^c(p,Q)$ seen so far. Their implementations $\mathtt{l}_i\mathtt{(p)}$ and $\mathtt{l}_i\mathtt{(p,q)}$ in Fig. 26 are counter objects, whose values are initialized to `0`. A counter object `C` provides functions `C.val` returning the current value and `C.incr` incrementing the current value by 1. Function `Counters.test` checks whether its argument $A, F \models \mathsf{f.frb}_i(p,q)$ holds, by comparing the current values of the counters $\mathtt{l}_i\mathtt{(p).val()} > \mathtt{l}_i\mathtt{(p,q).val()}$. Here, we use a Boolean valued function > for comparing integers.

```
// Inputs:
//   − A: productive stepwise tree automaton
//   − F: deterministic factorized tree automaton over the same signature
// Output: true iff L(A) ⊆ L(F)
fun inclusion(A,F)
  exception fail0 fail1 fail2
  ⟨⟨ create counters ⟩⟩
  ⟨⟨ create literal collection ⟩⟩
  ⟨⟨ create agenda with priorities ⟩⟩
  try
    // compute lfp(D2(A,F))
    // check for failures on the fly
    // raise exception once a failure condition gets valid
    ⟨⟨ saturate agenda with priorities ⟩⟩
    return true // no accessible states got forbidden!
  catch fail0 fail1 fail2 then
    return false
```

Fig. 25. Algorithm in pseudo-language for testing inclusion.

```
// create counters with initial value 0
forall p ∈ sta(A) do
   l1(p) = counter.new(0)
   l2(p) = counter.new(0)
   forall q ∈ sta(F) do
      l1(p,q) = counter.new(0)
      l2(p,q) = counter.new(0)
// check membership of f.frb_i literals to the least fixed point
fun Counters.test(lit)
  case lit
  of f.frb1(p,q) then
    return l1(p).val() > l1(p,q).val()
  of f.frb2(p,q) then
    return l2(p).val() > l2(p,q).val()
```

Fig. 26. ⟨⟨ create counters ⟩⟩

```
let heap = Set.new(∅) in // initialize collection of literals
fun Literals.mem(lit)   // test membership of literal to collection
  return heap.mem(lit)
proc Literals.add(lit)  // add literal to collection
  case lit
  of acc(p,q) then
    if not Literals.mem(acc(p,q)) then
      if Literals.mem(frb(p,q)) then
        raise fail_1 // apply (fail_1/a)
      if exists r ∈ sta(F) such that q -ε->_F r then
        if Literals.induced(f.frb_sort(q)(p,q))
           or Literals.induced(f.frb_sort(r)(p,r)) then
          raise fail_1 // early failure detection
        if p ∈ fin(A) and q ∉ fin(F) and r ∉ fin(F) then
          raise fail_2 // apply (fail_2/a)
      else // ∄ r ∈ sta(F) such that q -ε->_F r
        ... // same as above but without r
      heap.put(acc(p,q))
      Agenda.put_low(acc(p,q))
  of f.acc(p,q) then
    if not Literals.mem(f.acc(p,q)) then
      heap.put(f.acc(p,q))
      Agenda.put_high(f.acc(p,q))
  of acc(p,_) then
    if not Literals.mem(acc(p,_)) then
      heap.put(acc(p,_))
      Agenda.put_low(acc(p,_))
  of frb(p,q) then
    if not Literals.mem(frb(p,q)) then
      if Literals.mem(acc(p,q)) then
        raise fail_1 // apply (fail_1/a)
      heap.put(frb(p,q))
      Agenda.put_low(frb(p,q))
  of f.frb_i^c(p,Q) then
    Agenda.put_high(f.frb_i^c(p,Q))
```

Fig. 27. ⟨⟨ create literal collection ⟩⟩

```
let high = Stack.new(∅) in // define a higher priority stack
let low = Stack.new(∅)  in // define a lower priority stack
// interface
fun Agenda.nonempty()
  return high.nonempty() and low.nonempty()
fun Agenda.nonempty_high()
  return high.nonempty()
fun Agenda.nonempty_low()
  return low.nonempty()
fun Agenda.get_high()
  return high.pop()
fun Agenda.get_low()
  return low.pop()
proc Agenda.put_high(literal)
  high.push(literal)
proc Agenda.put_low(literal)
  low.push(literal)
```

Fig. 28. ⟨⟨ create agenda with priorities ⟩⟩

Object `Literals` in Fig. 27 collects all literals inferred so far, with the exception of frb_i^c literals. Function `Literals.mem` tests membership of its argument to this collection. Procedure `Literals.add` adds a literal if it is not yet present in the collection and applies all clauses to it. For literals $\mathsf{acc}(p, q)$, one first checks whether $\mathtt{fail}_1$ should be raised, either because $\mathsf{frb}(p, q)$ has been inferred before or due to some implied $\mathsf{f.frb}_i$ literal (early detection of failure $\mathtt{fail}_1$). Second, $\mathtt{fail}_2$ is checked according to (fail_2). Third, the literal is put to the collection and onto the agenda with low priority (see below). The addition of literals $\mathsf{frb}(p, q)$ is successful if $\mathsf{acc}(p, q)$ has not been added before. Otherwise exception $\mathtt{fail}_1$ is raised. New literals $\mathsf{f.acc}$ are added to the collection and the agenda with high priority. New literals $\mathsf{acc}(p, \_)$ are treated the same way, except that they are put to the agenda with low priority.

Predicates $\mathsf{f.frb}_i^c$ are always put to the agenda with high priority without further checking. Note that, by this, we permit the addition of the same literal several times. The only operation with those literals will be to increment counters. Also, no

```
// schedule literals for constant rules
forall a,p such that a → p ∈ rul(A) do
  if exists q ∈ sta(F) such that a → q ∈ rul(F) then
    Literals.add(acc(p,q)) // apply (acc/1a)
  else
    raise fail0 // apply (fail0/a)
// saturate agenda with priorities
while Agenda.nonempty() do
  while Agenda.nonempty_high() do
    ⟨⟨ apply rules with higher priority ⟩⟩
  if Agenda.nonempty_low() then
    ⟨⟨ apply rules with lower priority ⟩⟩
```

Fig. 29. ⟨⟨ saturate agenda with priorities ⟩⟩

```
case Agenda.get_high()
of f.acc(p,q) then
  case sort(q)
  of 1 then
    let p1 = p in
      forall p2 such that A⊨p1@p2 do
        Literals.add(f.frb^c_2(p2,Q^F_2(q))) // apply (f.frb^c_2)
  of 2 then
    ... // apply (f.frb^c_1) symmetrically to 1
  Agenda.put_low(f.acc(p,q)) // schedule for low priority operations
of f.frb^c_i(p,Q) then
  l_i(p).incr()
  forall q ∈ Q do
    l_i(p,q).incr()
```

Fig. 30. ⟨⟨ apply rules with higher priority ⟩⟩

```
case Agenda.get_low()
of acc(p,q) then
  forall r such that q →^{ε ≤1}_F r do
    Literals.add(f.acc(p,r)) // apply (f.acc)
  forall p' such that p →^ε_A p' do
    Literals.add(acc(p',q))  // apply (acc/2a)
  Literals.add(acc(p,_))     // apply (acc/4)
of acc(p,_) then
  // apply (frb/2a)
  let p1 = p in
    forall p2 such that A⊨p1@p2 do
      forall q2 ∉ R^F_2 do
        Literals.add(frb(p2,q2))
  // apply (frb/1a)
  ... // symmetric to (frb/2a)
of f.acc(p,q) then
  case sort(q)
  of 1 then
    let (p1,q1) = (p,q) in
      forall p2,p' such that p1@p2 → p' ∈ rul(A) do
        forall q2,q' such that q1@q2 → q' ∈ rul(F) do
          if Literals.mem(f.acc(p2,q2)) then
            Literals.add(acc(p',q')) // apply (acc3/a)
  of 2 then
    ... // symmetric to 1
```

Fig. 31. ⟨⟨ apply rules with lower priority ⟩⟩

rule for those predicates implies the inference of other literals to the fixed point. The halt of the saturation process described below only depends on acc and f.acc predicates. Since such literals are added at most once, halt is guaranteed.

We use an object `Agenda` defined in Fig. 28 that stores all literals to which some clauses remain to be applied. Literals in the agenda are either tagged with priority `high` or `low`, as required for early detection of failure $\mathtt{fail}_1$. High priority literals may serve in clauses that produce literals of high priority only. The first addition of a literal to the agenda is always with `high`. Once all consequences of high priority are produced, the tag is changed to `low`. Here, we optimize the previous processing by noticing that low priority suffices for treating acc and frb literals, as well as $\mathsf{f.frb}^c_i$ literals only have high priorities. The functions of object `Agenda` are `nonempty()`, `get_high()`, `get_low()`, `put_high(lit)`, and `put_low(lit)`.

After initializations, the algorithm starts the saturation process of Fig. 29. In a first time, it applies the rule $(\mathrm{acc}_{/1a})$ for adding accessible states of $A \times F$ from constants. In parallel it tests for failure $\mathtt{fail}_0$ and raises the appropriate exception if the rule $(\mathrm{fail}_{0/a})$ can be applied. During this process, the agenda is filled with low priority acc literals.

In a second step, the saturation procedure loops on the agenda and applies the priority policy as follows. It first applies all the rules for literals with the highest priority until it is empty. Then it applies the rules for a unique literal with low priority. This step can indeed involve the scheduling of tasks with a higher priority. The loop stops whenever the whole agenda is empty, or if some exception is raised during saturation.

Applying rules with high priority is described in Fig. 30. It does not concern acc literals as previously stated. For f.acc literals, the highest priority is to infer $\mathrm{f.frb}_i^c$ literals with respect to sorts $i \in \{1, 2\}$ by applying $(\mathrm{f.frb}_1^c)$ and $(\mathrm{f.frb}_2^c)$ rules of Fig. 15. Then the literal is scheduled for low priority operations. For literals $\mathrm{f.frb}_i^c(p, Q)$, counter $\mathtt{l}_i\mathtt{(p)}$ and counters $\mathtt{l}_i\mathtt{(p,q)}$ have to be incremented for all $q \in Q$ (see Lemma 24). This way, multiple occurrences of such literals in the least fixed point are properly taken into account, as previously noticed.

Applying rules with low priority is described in Fig. 31. It concerns only literals acc and f.acc. For both, the job consists in verifying the necessary conditions to apply all the rules where they appear in the tail of a Datalog clause, namely (f.acc), $(\mathrm{acc}_{/2a})$, $(\mathrm{acc}_{/4})$, $(\mathrm{frb}_{/1a})$ and $(\mathrm{frb}_{/2a})$ for acc literals, and only $(\mathrm{acc}_{/3a})$ for f.acc since other rules are applied with high priority.

References

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, available from: <http://www.grappa.univ-lille3.fr/tata/>, 1997, extended and revised in October 2007.
[2] H. Seidl, Deciding equivalence of finite tree automata, SIAM Journal on Computing, 19 (3) (1990) 424–437.
[3] H. Seidl, Haskell overloading is DEXPTIME-complete, Information Processing Letters 52 (2) (1994) 57–60.
[4] A. Neumann, H. Seidl, Locating matches of tree patterns in forests, in: International Conference on Foundations of Software Technology and Theoretical Computer Science (FCTTCS), 1998, pp. 134–145.
[5] S.D. Zilio, D. Lugiez, XML Schema, tree logic and sheaves automata, in: International Conference on Rewriting Techniques and Applications (RTA), 2003, pp. 246–263.
[6] S. Maneth, A. Berlea, T. Perst, H. Seidl, XML type checking with macro tree transducers, in: Symposium on Principles of Database Systems (PODS), 2005, pp. 283–294.
[7] T. Schwentick, Automata for XML—a survey, Journal of Computer and System Science 73 (3) (2007) 289–315.
[8] T. Milo, D. Suciu, V. Vianu, Type checking XML transformers, Journal of Computer and System Science 1 (66) (2003) 66–97.
[9] J. Champavère, R. Gilleron, A. Lemay, J. Niehren, Schema-guided induction of monadic queries, in: International Colloquium on Grammatical Inference (ICGI), 2008, pp. 15–28.
[10] J. Carme, J. Niehren, M. Tommasi, Querying unranked trees with stepwise tree automata, in: International Conference on Rewriting Techniques and Applications (RTA), 2004, pp. 105–118.
[11] G.J. Bex, F. Neven, J.V. den Bussche, DTDs versus XML Schema: a practical study, in: International Workshop on the Web and Databases (WebDB), 2004, pp. 79–84.
[12] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Information and Computation 142 (2) (1998) 182–206.
[13] W. Martens, J. Niehren, On the minimization of XML schemas and tree automata for unranked trees, Journal of Computer and System Science 73 (4) (2007) 550–583.
[14] S. Raeymaekers, Information extraction from web pages based on tree automata induction, Ph.D. Thesis, Katholieke Universiteit Leuven (January 2008).
[15] J.W. Thatcher, Characterizing derivation trees of context-free grammars through a generalization of automata theory, Journal of Computer and System Science 1 (1967) 317–322.
[16] A. Brüggemann-Klein, D. Wood, M. Murata, Regular tree and regular hedge languages over unranked alphabets: Version 1 (April 07 2001). Available from: <http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz/>.
[17] W. Martens, F. Neven, T. Schwentick, Which XML schemas admit 1-pass preorder typing?, in: International Conference on Database Theory, 2005, pp. 68–82.
[18] W. Martens, F. Neven, T. Schwentick, G.J. Bex, Expressiveness and complexity of XML Schema, ACM Transactions on Database Systems 31 (3) (2006) 770–813.
[19] E. van der Vlist, XML Schema, O'Reilly Media, Inc., 2002.
[20] E. van der Vlist, Relax NG, O'Reilly Media, Inc., 2003.
[21] J. Champavère, R. Gilleron, A. Lemay, J. Niehren, Efficient inclusion checking for deterministic tree automata and DTDs, in: International Conference on Language and Automata Theory and Applications (LATA), 2008, pp. 184–195.
[22] W. Martens, F. Neven, T. Schwentick, Complexity of decision problems for simple regular expressions, in: International Symposium on Mathematical Foundations of Computer Science (MFCS), 2004, pp. 889–900.
[23] A. Tozawa, M. Hagiya, XML Schema containment checking based on semi-implicit techniques, in: International Conference on Implementation and Application of Automata (CIAA), 2003, pp. 51–61.
[24] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about Datalog (and never dared to ask), IEEE Transactions on Knowledge Data Engineering 1 (1) (1989) 146–166.
[25] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Computing Surveys 33 (3) (2001) 374–425.
[26] G. Gottlob, E. Grädel, H. Veith, Datalog LITE: a deductive query language with linear time model checking, ACM Transactions on Computational Logics 3 (1) (2002) 42–79.
[27] M. Minoux, LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation, Information Processing Letters 29 (1) (1988) 1–12.
[28] A. Brüggemann-Klein, Regular expressions to finite automata, Theoretical Computer Science 120 (2) (1993) 197–213.
[29] G. Gottlob, C. Koch, Monadic queries over tree-structured data, in: IEEE Symposium on Logic in Computer Science (LICS), 2002, pp. 189–202.
[30] J. Viràgh, Deterministic ascending tree automata I, Acta Cybernetica 5 (1) (1980) 33–42.
[31] M. Murata, D. Lee, M. Mani, Taxonomy of XML schema languages using formal language theory, ACM Transactions on Internet Technology 5 (4) (2005) 660–704.
[32] O. Gauwin, J. Niehren, Y. Roos, Streaming tree automata, Information Processing Letters 109 (1) (2008) 13–17.
[33] R. Alur, Marrying words and trees, in: Symposium on Principles of Database Systems (PODS), 2007, pp. 233–242.