Simplifying XML Schema: Effortless Handling of Nondeterministic Regular Expressions[*]

Geert Jan Bex
Hasselt University and Transnational University of Limburg
geertjan.bex@uhasselt.be

Wouter Gelade[†]
Hasselt University and Transnational University of Limburg
wouter.gelade@uhasselt.be

Wim Martens[‡]
Technical University of Dortmund
wim.martens@udo.edu

Frank Neven
Hasselt University and Transnational University of Limburg
frank.neven@uhasselt.be

ABSTRACT

Whether beloved or despised, XML Schema is momentarily the only industrially accepted schema language for XML and is unlikely to become obsolete any time soon. Nevertheless, many nontransparent restrictions unnecessarily complicate the design of XSDs. For instance, complex content models in XML Schema are constrained by the infamous unique particle attribution (UPA) constraint. In formal language theoretic terms, this constraint restricts content models to deterministic regular expressions. As the latter constitute a semantic notion and no simple corresponding syntactical characterization is known, it is very difficult for non-expert users to understand exactly when and why content models do or do not violate UPA. In the present paper, we therefore investigate solutions to relieve users from the burden of UPA by automatically transforming nondeterministic expressions into concise deterministic ones defining the same language or constituting good approximations. The presented techniques facilitate XSD construction by reducing the design task at hand more towards the complexity of the modeling task. In addition, our algorithms can serve as a plug-in for any model management tool which supports export to XML Schema format.

[*]We acknowledge the financial support of FWO-G.0821.09N and the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599

[†]Research Assistant of the Fund for Scientific Research – Flanders (Belgium).

[‡]Supported by the North-Rhine Westphalian Academy of Sciences, Humanities and Arts; and the Stiftung Mercator Essen.

Categories and Subject Descriptors

F.4.3 [**Theory of Computation**]: Mathematical Logic and Formal Languages—*Formal Languages*; I.7.2 [**Computing Methodologies**]: Document and Text Processing—*Document Preparation*

General Terms

Algorithms Languages Theory

Keywords

XML Schema, UPA, deterministic regular expressions

1. INTRODUCTION

The presence of a schema accompanying an XML repository has many advantages: (*i*) it strongly facilitates optimization of XML processing (cf., e.g., [3, 11, 13, 15, 22, 23, 27]); (*ii*) it provides a road map for the user to the underlying data; and (*iii*) it is inevitable when integrating (meta) data through schema matching [28] and in the area of generic model management [4]. Despite these many advantages, recent studies stipulate that schemas accompanying collections of XML documents are sparse and erroneous in practice: Barbosa et al. [2, 26] have shown that approximately half of the XML documents available on the Web do not refer to a schema; Bex et al. [5, 25] noted that about two-thirds of XML Schema Definitions (XSDs) gathered from schema repositories and from the Web at large are not valid with respect to the W3C XML Schema specification [30], rendering them essentially useless for immediate application. Although the exact causes of the absence of schemas and the high percentage of errors in XSDs are difficult to pinpoint, the high complexity of XML Schema itself undoubtedly plays an important role, distracting the attention from the design process itself. We therefore need easy-to-use schema design tools reducing the design task solely to the complexity of modeling by relieving the user from XML Schema's peculiarities and nontransparent restrictions.

One of these nontransparent restrictions is undoubtedly the Unique Particle Attribution (UPA) constraint, also known as the restriction on content models to be *deterministic* regular expressions (as formally defined in Section 2). The sole motivation for this restriction is backward compatibility with SGML, a predecessor of XML, where it was introduced for the reason of fast unambiguous parsing of content models (without lookahead) [31]. Sadly this notion of unambiguous parsing is a semantic rather than a syntactic one, making it difficult for designers to interpret. Specifically, the XML Schema specification mentions the following definition of UPA:

> A content model must be formed such that during *validation* of an element information item sequence, the particle component contained directly, indirectly or *implicitly* therein with which to attempt to *validate* each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.

In most books (c.f. [31]), the UPA constraint is usually explained in terms of a simple example rather than by means of a clear syntactical definition. The latter is not surprising as to date there is no known easy syntax for deterministic regular expressions. That is, there are no simple rules a user can apply to define only (and all) deterministic regular expressions. So, when after the schema design process, one or several content models are rejected by the schema checker on account of being nondeterministic, it is very difficult for non-expert[1] users to grasp the source of the error and almost impossible to rewrite the content model into an admissible one. The purpose of the present paper is to investigate methods for transforming nondeterministic expressions into concise and readable deterministic ones defining either the same language or constituting good approximations. We propose the algorithm SUPAC (Supportive UPA Checker) which can be incorporated in a responsive XSD tester which in addition to rejecting XSDs violating UPA also suggests plausible alternatives. Consequently, the task of designing an XSD is relieved from the burden of the UPA restriction and the user can focus on designing an accurate schema. In addition, our algorithm can serve as a plug-in for any model management tool which supports export to XML Schema format [4].

Deterministic regular expressions were investigated in a seminal paper by Brüggemann-Klein and Wood [10]. They showed that deciding whether a given regular expression is deterministic can be done in quadratic time. In addition, they provided an algorithm, that we call BKW$^{\text{dec}}$, to decide whether a regular language can be represented by a deterministic regular expression.[2] BKW$^{\text{dec}}$ runs in time quadratic in the size of the minimal deterministic finite automaton (DFA) and therefore in time exponential in the size of the regular expression. We prove in this paper that the problem is hard for PSPACE thereby eliminating much of the hope for a theoretically tractable algorithm. We tested BKW$^{\text{dec}}$ on a large and diverse set of regular expressions and observed that it runs very fast (under 200ms for expressions with 50 alphabet symbols). It turns out that, for many expressions, the corresponding minimal DFA is quite small and far from the theoretical worst-case exponential size increase. In addition, we observe that BKW$^{\text{dec}}$ is fixed-parameter tractable in the maximal number of occurrences of the same alphabet symbol. As this number is very small for the far majority of real-world regular expressions [7], applying BKW$^{\text{dec}}$ in practice should never be a problem.

Deciding existence of an equivalent deterministic regular expression or effectively constructing one, are entirely different matters. Indeed, while the decision problem is in EXPTIME, Brüggemann-Klein and Wood [10] provide an algorithm, which we will call BKW, which constructs deterministic regular expressions whose size can be doubly exponential in the size of the original expression. In this paper, we measure the size of an expression as the total number of occurrences of alphabet symbols (cf. Section 2). The first exponential size increase stems from creating the minimal deterministic automaton A_r equivalent to the given nondeterministic regular expression r. The second one stems from translating the automaton into an expression. Although it is unclear whether this doubly exponential size increase can be avoided, examples are known for which a single exponential blow-up is necessary [10]. We define an optimized version of BKW, called BKW-OPT, which optimizes the second step in the algorithm and can produce exponentially smaller expressions than BKW. Unfortunately, the obtained expressions can still be very large. For instance, as detailed in the experiments section, for input expressions of size 15, BKW and BKW-OPT generate equivalent deterministic expressions of average size 1577 and 394, respectively. To overcome this, we propose the algorithm GROW. The idea underlying this algorithm is that small deterministic regular expressions correspond to small Glushkov automata [10]: indeed, every deterministic regular expression r can be translated in a Glushkov automaton with as many states as there are alphabet symbols in r. Therefore, when the minimal automaton A_r is not Glushkov, GROW tries to extend A_r such that it becomes Glushkov. To translate the Glushkov automaton into an equivalent regular expression, we use the existing algorithm REWRITE [7]. Our experiments show that when GROW succeeds in finding a small equivalent deterministic expression its size is always roughly that of the input expression. In this respect, it is greatly superior to BKW and BKW-OPT. Nevertheless, its success rate is inversely proportional to the size of the input expression (we refer to Section 6.2 for details).

Finally, we focus on the case when no equivalent deterministic regular expression can be constructed for a given nondeterministic regular expression and an adequate super-approximation is needed. We start with a fairly negative result: we show that there is no smallest super-approximation of a regular expression r within the class of deterministic regular expressions. That is, whenever $L(r) \subsetneq L(s)$, and s is deterministic, then there is a deterministic expression s' with $L(r) \subsetneq L(s') \subsetneq L(s)$. We therefore measure the proximity between r and s relative to the strings up to a fixed length. Using this measure we can compare the quality of different approximations. We consider three algorithms. The first one is an algorithm of Ahonen [1] which essentially repairs BKW by adding edges to the minimal DFA whenever it gets stuck. The second algorithm operates like the first one but utilizes GROW rather than BKW to generate the

[1]In formal language theory.

[2]We call such regular languages *deterministic regular languages*.

corresponding deterministic regular expression. The third algorithm, called SHRINK, merges states, thereby generalizing the language, until a regular language is obtained with a corresponding concise deterministic regular expression. For the latter, we again make use of GROW, and of the algorithm KOA-TO-KORE of [6] which transforms automata to concise regular expressions. In our experimental study, we show in which situation which of the algorithms works best.

Conributions. The contributions of this paper can be listed as follows:

1. We prove theoretical intractability for deciding whether a regular expression defines a deterministic language. A further analysis shows fixed-parameter tractability which explains the surprisingly good performance of BKW$^{\text{dec}}$ in practice.
2. In addition to revisiting known algorithms for handling deterministic regular expressions like BKW and AHONEN-BKW, we propose several new ones: BKW-OPT, GROW, AHONEN-GROW, and SHRINK.
3. We prove that in general there exists no best approximation of a regular expression within the class of deterministic regular expressions. We then propose a proximity measure to assess the quality of an approximation.
4. Based on a detailed experimental analysis, we asses the merits of each of these algorithms in terms of the quality and the conciseness of computed expressions. Interestingly, crudely flavored methods like GROW and SHRINK outperform more sophisticated ones like BKW-OPT and AHONEN-BKW on all but the smallest of expressions.
5. Based on the experimental assessment, we propose the algorithm SUPAC (supportive UPA checker) for handling deterministic regular expressions. SUPAC is hence a combination of the different proposed algorithms.

Outline. The outline of the paper is as follows. In Section 2, we introduce the necessary definitions. In Section 3, we discuss the complexity of deciding determinism of the underlying regular language. In Section 4 and Section 5, we discuss the construction of equivalent and super approximations of regular expressions. We present an experimental validation of our algorithms in Section 6. We outline a supportive UPA checker in Section 7. We conclude in Section 8.

Related Work. Although XML is accepted as the de facto standard for data exchange on the Internet and XML Schema is widely used, fairly little attention has been devoted to the study of deterministic regular expressions. We already mentioned the seminal paper by Bruggemann-Klein and Wood [10]. Computational and structural properties were addressed by Gelade and Neven [16] and Martens, Neven, and Schwentick [24]. In particular, it was shown that deterministic regular expressions can be complemented in polynomial time (although the complement is not necessarily deterministic [16]). Furthermore, it was shown that obtaining a regular expression for the intersection of an arbitrary number of deterministic expressions cannot avoid a doubly exponential size increase [16]. In addition, testing non-emptiness of an arbitrary number of intersections of deterministic regular expressions is PSPACE-complete [24]. Bex et al. investigated algorithms for the inference of regular expressions from a sample of strings in the context of DTDs and XML Schemas [6, 7, 8, 9]. From this investigation resulted two algorithms: REWRITE [7] which transforms automata with n states to equivalent expressions with n alphabet symbols and fails when no such expression exists; and the algorithm KOA-TO-KORE [6, 7] which operates as REWRITE with the difference that it always returns a concise expression at the expense of generalizing the language when no equivalent concise expression exists.

Deciding determinism of expressions containing numerical occurrences was studied by Kilpeläinen and Tuhkanen [20]. The complexity of syntactic subclasses of the deterministic regular expressions with counting has also been considered [17, 18]. In the context of streaming the notion of determinism and k-determinism was used in [21] and [12].

The presented work would clearly benefit from algorithms for regular expression minimization. To the best of our knowledge, no such (efficient) algorithms exist for deterministic regular expressions, for which minimization is in NP. Only a sound and complete rewriting system is available for general regular expressions [29], for which minimization is PSPACE-complete.

2. DEFINITIONS

In this paper, we denote by Σ an arbitrary finite alphabet, and by $a, b, c, \ldots$ we denote elements from Σ. We are interested in regular expressions r of the form

$$r ::= \varepsilon \mid \emptyset \mid a \mid rr \mid r + r \mid (r)? \mid (r)^{+} \mid (r)^{*},$$

where ε denotes the empty string and a ranges over symbols in the alphabet Σ. Sometimes we also use the symbol $\cdot$ for regular expression concatenation to improve readability. As usual, we write $L(r)$ for the language defined by regular expression r. We define the *size* of regular expression r to be its total number alphabet symbol occurrences. For example, both expressions aaa and $a(b+c)?$ have size three. We say that two regular expressions r_1 and r_2 are *equivalent* if $L(r_1) = L(r_2)$.

XML Schema does not allow the general class of regular expressions as defined above but requires regular expressions to be *deterministic* (also sometimes called one-unambiguous [10]). Intuitively, a regular expression is deterministic if, without looking ahead in the input string, it allows to match each symbol of that string uniquely against a position in the expression when processing the input in one pass from left to right. For instance, $(a + b)^*a$ is not deterministic as already the first symbol in the string aaa could be matched by either the first or the second a in the expression. Without lookahead, it is impossible to know which one to choose. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic. Formally, let $\bar{r}$ stand for the regular expression obtained from r by replacing the i-th occurrence of alphabet symbol a in r by a_i, for every i and a. For example, for $r = b^*a(b^*a)^*$ we have $\bar{r} = b_1^*a_1(b_2^*a_2)^*$.

DEFINITION 1. *A regular expression r is* deterministic *if there are no strings wa_iv and wa_jv' in $L(\bar{r})$ such that $i \neq j$.*

Equivalently, an expression is deterministic if the Glushkov construction translates it into a deterministic finite automaton rather than a non-deterministic one [10]. Furthermore, not every nondeterministic regular expression is equivalent to a deterministic one [10]. Thus, semantically, the class of deterministic regular expressions forms a strict subclass of the class of all regular expressions. We call a regular language *deterministic* if there exists a deterministic regular expression defining it.

Finite automata will be written as tuples $A = (Q, \Sigma, \delta, q_0, F)$, where Q is its set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation (where $(q_1, a, q_2) \in \delta$ means that A can enter state q_2 when reading an a in state q_1), q_0 is the initial state, and F is the set of final states. For a transition (q_1, a, q_2) we refer to q_1 as the *source state* and q_2 as the *target state*. A finite automaton is *complete* if, for every $(q, a) \in Q \times \Sigma$, there exists a q' such that $(q, a, q') \in \delta$. It is *deterministic* if, for every $(q, a) \in Q \times \Sigma$, there exists at most one q' such that $(q, a, q') \in \delta$. We denote the class of (non-deterministic) finite automata by NFA and the class of deterministic finite automata by DFA.

We say that a set of states $Q' \subseteq Q$ is *strongly connected* if, for each pair of states $q_1, q_2 \in Q'$, q_1 is reachable from q_2. A strongly connected set is maximal if there exists no superset that is strongly connected. In the remainder of the paper, whenever we refer to *strongly connected components* of an NFA, we always mean *maximal strongly connected state sets*.

3. DECIDING DETERMINISM

The first step in creating a responsive UPA checker is testing whether $L(r)$ is deterministic. Brüggemann-Klein and Wood obtained an EXPTIME algorithm (in the size of the regular expression) which we will refer to as BKW$^{\text{dec}}$:

THEOREM 2 ([10]). *Given a regular expression r, the algorithm BKW^{dec} decides in time quadratic in the size of the minimal DFA corresponding to r whether $L(r)$ is deterministic.*

We show that, unless PSPACE = PTIME, there is no hope for a tractable algorithm. The proof of the following theorem is given in the appendix of this paper.

THEOREM 3. *Given a regular expression r, the problem of deciding whether $L(r)$ is deterministic is PSPACE-hard.*

It is unclear whether the problem itself is in PSPACE. Simply guessing a deterministic regular expression s and testing equivalence with r does not work as the size of s can be exponential in the size of r (see also Theorem 6).

Next, we address the problem from the viewpoint of parameterized complexity [14], where an additional parameter k is extracted from the input r. Then, we say that a problem is *fixed parameter tractable* if there exists a computable function f and a polynomial g such that the problem can be solved in time $f(k) \cdot g(|r|)$. Intuitively, this implies that, if k is small and f is reasonable, the problem is efficiently solvable. We now say that an expression r is a *k-occurrence regular expression (k-ORE)* if every alphabet symbol occurs at most k times in r. For example, $ab(a^* + c)$ is a 2-ORE because a occurs twice.

PROPOSITION 4. *Let r be a k-ORE. The problem of deciding whether $L(r)$ is deterministic is fixed parameter tractable with parameter k. Specifically, its complexity is $\mathcal{O}(2^{k^2}|r|^2)$.*

PROOF. Let r be a k-ORE. By applying a Glushkov construction (see, e.g., [10]), followed by a subset construction, it is easy to see that the resulting DFA has at most $2^k \cdot |\Sigma|$ states. By Theorem 2, the result follows. □

Algorithm 1 Algorithm GROW, with pool size P and expansion size E.

```
Input: $P$, $E \in \mathbb{N}$, minimal DFA $A = (Q, \Sigma, \delta, q_0, F)$ with
    $L(A)$ deterministic,
Output: Det. reg. exp. $s$ with $L(s) = L(A)$, if successful
    for $i = 0$ to $E$ do
 2:     Generate at most $P$ non-isomorphic DFAs $B$ s.t.
                         $L(B) = L(A)$ and $B$ has $|Q| + i$ states
 4:     for each such $B$ do
            if REWRITE ($B$) succeeds then
 6:             return REWRITE ($B$)
    if $i > E$ then fail
```

This result is not only of theoretical interest. It has already often been observed that the vast majority of regular expressions occurring in practice are k-OREs, for $k = 1, 2, 3$ (see, e.g., [25]). Hence, this result implies that in practice the problem can be solved in polynomial time.

COROLLARY 5. *For any fixed k, the problem of deciding whether the language defined by a k-ORE is deterministic is in PTIME.*

4. CONSTRUCTING DETERMINISTIC EXPRESSIONS

Next, we focus on constructing equivalent deterministic regular expressions. Unfortunately, the following result by Brüggemann-Klein and Wood already rules out a truly efficient conversion algorithm:

THEOREM 6 ([10]). *For any $n \in \mathbb{N}$, there exists a regular expression r_n of size $\mathcal{O}(n)$ such that any deterministic regular expression defining $L(r_n)$ is of size at least 2^n.*

4.1 Growing automata

We first present GROW as Algorithm 1, which is designed to produce concise deterministic expressions. The idea underlying this algorithm is that the Glushkov construction [10] transforms small deterministic regular expressions to small deterministic automata with as many states as there are alphabet symbols in the expression. The minimization algorithm eliminates some of these states, complicating the inverse Glushkov-rewriting from DFA to deterministic regular expression. By expanding the minimal automaton, GROW tries to recuperate the eliminated states. The algorithm REWRITE of [7] succeeds when the modified automaton can be obtained from a deterministic regular expression by the Glushkov construction and assembles this expression upon success. As their are many DFAs equivalent to the given automaton A, we only enumerate non-isomorphic expansions of A up to a given number of extra states E. Thereto, we implemented an algorithm which given A and E enumerates all such non-isomorphic DFAs equivalent to A in time linear in the output size. However, as the algorithm is a bit technical, the description will be given in the full version of this paper. Nonetheless, the number of generated non-isomorphic DFAs can explode quickly. Therefore, the algorithm is also given a given pool size P which restricts the number of DFAs of each size which are generated.

Nonwithstanding the harsh brute force flavor of GROW, we show in our experimental study that the algorithm can be quite effective.

Algorithm 2 The BKW-Algorithm.

```
Input: Minimal DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: Det. reg. exp. $s$ with $L(s) = L(A)$
    if $A$ has only one state $q$ and no transitions then
 2:   if $q$ is final then return $\varepsilon$
      else return $\emptyset$
 4: else if $A$ has precisely one orbit then
      $S \leftarrow A$-consistent symbols
 6:   if $S = \emptyset$ then fail
      else return $\text{BKW}(A_S) \cdot \big(\bigcup_{a \in S} a \cdot \text{BKW}(A_S^{w(a)})\big)^*$
 8: else
      if $A$ has the orbit property then
10:     for all $a$ s.t. Orbit($q_0$) has outgoing $a$-transition do
          $q_a \leftarrow$ unique target state of these $a$-transitions
12:     $A_{q_0} \leftarrow$ orbit automaton of $q_0$
        if $A_{q_0}$ contains a final state then
14:       return $\text{BKW}(A_{q_0}) \cdot \big(\bigcup_{a \in \Sigma}(a \cdot \text{BKW}(A^{q_a}))\big)?$
        else
16:       return $\text{BKW}(A_{q_0}) \cdot \bigcup_{a \in \Sigma}(a \cdot \text{BKW}(A^{q_a}))$
      else fail
```

4.2 Optimizing the BKW-Algorithm

Next, we discuss Brüggemann-Klein and Woods BKW algorithm and then present a few optimizations to generate smaller expressions.

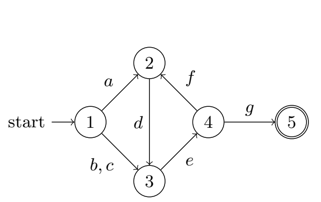
First, we need some terminology. Given a DFA A, a symbol a is *A-consistent* if there is a unique state $w(a)$ in A such that all final states of A have an a-transition to $w(a)$. We call $w(a)$ the *witness state* for a. A set S is A-consistent if each element in S is A-consistent. The *S-cut* of A, denoted by A_S, is the automaton obtained from A by removing, for each $a \in S$, all a-transitions that leave a final state of A. Given a state q of A, A^q is the automaton obtained from A by setting its initial state to q and restricting its state set to the states reachable from q. For a state q, the *orbit of* q, denoted Orbit(q), is the strongly connected component of A that contains q. We call q a *gate* of Orbit(q) if q is final, or q is the source of a transition that has a target outside Orbit(q).

We say that A has *the orbit property* if, for every pair of gates q_1, q_2 in the same orbit the following properties hold:

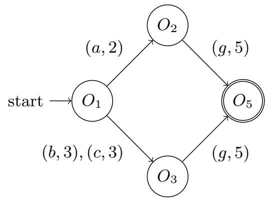
1. q_1 is final if and only if q_2 is final; and,
2. for all states q outside the orbit of q_1 and q_2, there is a transition (q_1, a, q) iff there is a transition (q_2, a, q).

Given a state q of A, the *orbit automaton* of q, denoted by A_q, is obtained from A by restricting its state set to Orbit(q), setting its initial state to q and by making the gates of Orbit(q) its final states.

The BKW-Algorithm is then given as Algorithm 2. For a regular expression r, the algorithm is called with the minimal complete DFA A accepting $L(r)$ and then recursively constructs an equivalent deterministic expression when one exists and fails otherwise. Algorithm 2 can fail in two places: (1) in line 6, when the set of A-consistent symbols is empty and (2) in line 17, if A does not have the orbit property. Notice that, if A has the orbit property, the unique state q_a on line 11 can always be found. The correctness proof is non-trivial and can be found in [10]. BKW runs in time doubly exponential in the size of the nondeterministic regular expression. The first exponential arises from converting to a DFA, the second one from branching in the lines 7, 14, and 16. The generated expressions can therefore be quite large. As Algorithm 2 was not designed with conciseness of regular expressions in mind, we therefore propose three optimizations resulting in smaller expressions.



(a) DFA A.



(b) summ(A)

Figure 1: A DFA and its summary automaton.

To this end, let first(A) denote the set $\{a \mid \exists w \in \Sigma^*, aw \in L(A)\}$, i.e., the set of possible first symbols in a string in $L(A)$. We adapt the lines 7, 14, and 16 in Algorithm 2 in the way described below and refer to the modified algorithm as BKW-OPT.

$\boxed{\textit{line 7}}$ Now, A consists of one orbit and S is the set of A-consistent symbols:

- If $L(A_S) = L(A_S^{w(a)})$ for all $a \in S$, $\varepsilon \in L(A_S)$, and first$(A_S) \cap S = \emptyset$, then return $((S + \varepsilon) \cdot \text{BKW}(A_S))^*$.
- Otherwise, partition S into equivalence classes $S_1, \ldots, S_n$ where for $a, b \in S$, a is equivalent to b iff $w(a) = w(b)$. Furthermore, let, for each $i \in \{1, \ldots, n\}$, a_i be an arbitrary but fixed element from S_i. Then, return $\text{BKW}(A_S) \cdot \big(\bigcup_{1 \le i \le n} S_i \cdot \text{BKW}(A_{S_i}^{w(a_i)})\big)^*$.

$\boxed{\textit{line 14 and 16}}$ If A consists of more than one orbit, we can view A as an acyclic DFA when considering every orbit as an atomic subautomaton. We therefore define the acyclic DFA summary automaton summ(A) of A where every state corresponds to a unique orbit. As these automata are usually quite small, we subsequently apply GROW to obtain a concise regular expression over an alphabet consisting of Σ-symbols and orbit identifiers. We then replace each orbit identifier by its corresponding, recursively obtained deterministic expression.

Before defining summary automata formally, we present an example. Figure 1(a) illustrates a DFA A with three orbits: $\{1\}$, $\{2,3,4\}$, and $\{5\}$. Orbit $\{2,3,4\}$ has two possible entry points: states 2 (with an a-transition) and 3 (with the b- and c-transitions). For each such entry point we have a state in the summary automaton. Figure 1(b) presents the summary automaton summ(A).

Formally, let $A = (Q, \Sigma, \delta, q_0, F)$. Then we define summ($A$) as a DFA $(Q^s, \Sigma^s, \delta^s, q_0^s, F^s)$, where $\Sigma^s \subseteq \Sigma \times Q$. In particular, for each transition $(q_1, a, q_2) \in \delta$ where Orbit$(q_1) \neq$ Orbit(q_2), we have $(a, q_2) \in \Sigma^s$. The state set Q^s is defined as $\{O_q \mid$ there is a transition $(p, a, q) \in \delta$ for p outside Orbit$(q)\}$. Furthermore, we define $q_0^s := O_{q_0}$, $F^s := \{O_p \in Q^s \mid \text{Orbit}(p) \cap F \neq \emptyset\}$, and $(O_{q_1}, (a, q_2), O_{q_2}) \in \delta^s$ iff Orbit$(q_1) \neq$ Orbit(q_2) and there exists a $q_1' \in$ Orbit(q_1) such that $(q_1', a, q_2) \in \delta$. Notice that, if A is a DFA fulfilling the orbit property, all outgoing transitions of each orbit go to the same witness state. Therefore, summ(A) is also a DFA.

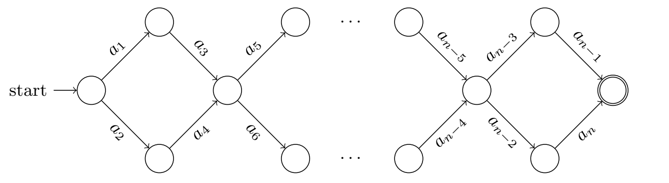


Figure 2: Class of DFAs for which our optimization improves exponentially over the BKW algorithm.

To find a small regular expression for the multiple orbits case, we make use of the deterministic expressions r_q for the orbit automata A_q that we computed deeper in the recursion. We run GROW on summ(A) to find a small deterministic expression for $L(\text{summ}(A))$. If we find one, we obtain the deterministic expression for $L(A)$ by replacing each symbol (a, q) by $a \cdot r_q$.

Notice that this optimization potentially generates exponentially smaller regular expressions than the BKW algorithm. Consider the family of DFAs of Figure 2. The summary DFAs for these automata are equal to the DFAs themselves. While the BKW algorithm would essentially unfold this DFA and return a regular expression of size at least 2^n, GROW would return the expression $(a_1a_3+a_2a_4)\cdots(a_{n-3}a_{n-1} + a_{n-2}a_n)$, which is linear in n.

It is shown in [19] that there are acyclic DFAs whose smallest equivalent regular expression of superpolynomial size: $\Omega(n^{\log n})$ for n the size of the DFA. As acyclic DFAs define finite languages and finite languages are deterministic, the result transfers to deterministic regular expressions. Therefore, when GROW does not find a small solution we just apply one non-optimized step of the BKW algorithm (i.e., return line 14/16 of Algorithm 2). However, in our experiments we noticed that this almost never happened (less than 1% of the total calls to GROW did not return an expression).

5. APPROXIMATING DETERMINISTIC REGULAR EXPRESSIONS

When the regular language under consideration is not deterministic, we can make it deterministic at the expense of generalizing the language. First, we show that there is no best approximation.

5.1 Optimal Approximations

An expression s is a *deterministic super-approximation* of an expression r when $L(r) \subseteq L(s)$ and s is deterministic. In the sequel we will just say approximation rather than super-approximation. Then, we say that s is an *optimal deterministic approximation* of r, if $L(r) \subseteq L(s)$, and there does not exist a deterministic regular expression s' such that $L(r) \subseteq L(s') \subsetneq L(s)$. That is, an approximation is optimal if there does not exist another one which is strictly better. Unfortunately, we can show that no such optimal approximation exists:

THEOREM 7. *Let r be a regular expression, such that $L(r)$ is not deterministic. Then, there does not exist an optimal deterministic approximation of r.*

PROOF. We show in Lemma 9 (shown in the Appendix) that for every deterministic language L and string $w \in L$, the language $L \setminus \{w\}$ is also deterministic. Now, suppose, towards a contradiction, that an optimal deterministic approximation s exists. Then, since $L(r)$ is not deterministic, $L(r) \subsetneq L(s)$ and thus there exists some string w with $w \in L(s)$ but $w \notin L(r)$. But then, for the language $L_w = L(s) \setminus \{w\}$, we have that $L(r) \subseteq L_w$ and, by Lemma 9, L_w is also deterministic. This gives us the desired contradiction. ☐

As finite languages are always deterministic, Theorem 7 implies that every approximation defines infinitely more strings than the original expression. Furthermore, one can prove analogously (using Lemma 10) that an optimal *under-approximation* of a non-deterministic regular expression r also does not exist. That is, a deterministic regular expression s such that $L(s) \subseteq L(r)$ for which there is no deterministic s' with $L(s) \subsetneq L(s') \subseteq L(r)$.

5.2 Quality of the approximation

Motivated by the above discussion, we will compare sizes of regular languages by only comparing strings up to a predefined length.

Thereto, for an expression r and a natural number ℓ, let $L^\ell(r)$ be the subset of strings in $L(r)$ with length exactly ℓ. For regular expressions r and s with $L(r) \subseteq L(s)$, define the *proximity* between r and s, as

$$\text{proximity}(r, s) := \frac{1}{k} \sum_{\ell=1}^{k} \frac{|\,L^\ell(r)| + 1}{|\,L^\ell(s)| + 1}$$

for $k = \max\{2|r| + 1, 2|s| + 1\}$. The proximity is always a value between 0 and 1. When the proximity is close to 1, the size of the sets $L^\ell(s) \setminus L^\ell(r)$ is small, and the quality of approximation is excellent.

Although the above measure provides us with a tool to compare proximity of regular languages, we cannot simply search for a deterministic expression which performs best under this measure. Indeed, there always is a deterministic expression s for which proximity(r, s) equals one. For instance, the language which contains every string of length larger than k and only those strings of smaller length which are in $L(r)$ is a deterministic language and can therefore be represented by a deterministic expression s. Clearly, such an expression will never be an acceptable approximation. Furthermore, its size grows (in general) exponentially in $|r|$.

In conclusion, a valid approximation s for an expression r, is a deterministic expression constituting a good tradeoff between (1) a large value for proximity(r, s) and (2) a small size $|s|$. The heuristics in the following section will try to construct approximations which fit these requirements.

5.3 Ahonen's Algorithm

In the previous section, we have seen that the BKW-algorithm will translate a DFA into a deterministic expression, and will fail if no such equivalent deterministic expression exists. Ahonen's algorithm [1] is a first method that constructs a deterministic regular expression at the expense of generalizing the target language. It essentially runs the BKW$^{\text{dec}}$-algorithm, the decision variant of BKW which does not produce an output expression (cf. Theorem 2), until it fails, and subsequently *repairs* the DFA by adding transitions or making states final, in such a manner that BKW$^{\text{dec}}$ can continue. In the end, a DFA is produced defining a deterministic language. The corresponding deterministic regular expression is then obtained by running BKW. Ahonen's algorithm for

Algorithm 3 An adaptation of Ahonen's repair algorithm: the AHONEN algorithm.

```
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: DFA $B$ such that $L(A) \subseteq L(B)$
    $S \leftarrow A$-consistent symbols
 2: if $A$ has only one state $q$ then
        if $q$ is final then return $\varepsilon$
 4:     else return $\emptyset$
    else if $A$ has precisely one orbit then
 6:     if $S = \emptyset$ then
            Choose an $a$ s.t. $(q, a, q_1) \in \delta$ for $q$ final
 8:         for all $p \in F$ do
                add $(p, a, q_1)$ to $\delta$
10:             if $(p, a, q_2) \in \delta$ for $q_2 \neq q_1$ then
                    Merge($q_1$,$q_2$)
12:         $S \leftarrow \{a\}$
    ForceOrbitProperty $(A_S)$
14: for each orbit $O$ of $A_S$ do
        Choose an arbitrary $q \in O$
16:     AHONEN $((A_S)_q)$
```

obtaining a DFA defining a deterministic language is presented in Algorithm 3.[3] By AHONEN-BKW we then denote the application of BKW on the result of AHONEN.

AHONEN proceeds by merging states. We explain in more detail how we can merge two states in a DFA $A = (Q, \Sigma, \delta, q_0, F)$. For an example, see Figure 3(c) and 3(d), where states 2 and 4 are merged into a new state $\{2, 4\}$. For ease of exposition, we assume that states in Q are sets. Initially, all sets in Q are singletons (e.g., $\{2\}$, $\{4\}$) and by merging such states we obtain non-singletons (e.g., $\{2, 4\}$). Let q_1 and q_2 be the two states to be merged into a new state $q_M := q_1 \cup q_2$. We denote by Q_{new} the state set of A after this merge operation (analogously for δ_{new},${q_0}_{\text{new}}$, and F_{new}). We assume that $q_1, q_2 \in Q$ and $q_M \notin Q$. Then, $Q_{\text{new}} := (Q \cup q_M) \setminus \{q_1, q_2\}$. Furthermore, $q_M \in F_{\text{new}}$ iff $q_1 \in F$ or $q_2 \in F$. Analogously, ${q_0}_{\text{new}}$ is the unique set $p \in Q_{\text{new}}$ such that $q_0 \in p$. The transitions will be adapted as follows: for all states $q_1, q_2 \in Q_{\text{new}}$ we have $(q_1, a, q_2) \in \delta_{\text{new}}$ iff there exist $q_1', q_2' \in Q$ such that $(q_1', a, q_2') \in \delta$. As long as the obtained automaton is not deterministic, we choose non-deterministic transitions (p, a, q_1) and (p, a, q_2) and continue merging states until it is deterministic again. We denote this recursive merging procedure in Algorithms 3 and 4 by Merge.

AHONEN repairs the automaton A in two possible instances where BKW$^{\text{dec}}$ gets stuck. If A has one orbit but no A-consistent symbols, AHONEN simply chooses a symbol a and adds transitions to force A-consistency. If A has more than one orbit, but does not fulfill the orbit property, then AHONEN calls ForceOrbitProperty to add final states and transitions until A fulfills it.

Example 8. *We illustrate the algorithm on the regular expression* $(aba + a)^+b$. *The minimal DFA A is depicted in Figure 3(a). As there are no A-consistent symbols, we have that* $S = \emptyset$. *As there are three orbits* ((1), (2, 3, 4), *and* (5)), *the algorithm immediately calls ForceOrbitProperty on A, where 4 is made final and transition* (3, b, 5) *is added*

[3]The algorithm we present slightly differs from Ahonen's original algorithm as the original algorithm is slightly incorrect. We briefly discuss this in the Appendix.

Algorithm 4 The ForceOrbitProperty procedure.

```
Input: DFA $A$
Output: DFA $B$ such that $L(A) \subseteq L(B)$
    for each orbit $O$ of $A$ do
18:     Let $g_1, \ldots, g_k$ be the gates of $K$
        if there exists a gate $g_i \in F$ then
20:         $F \leftarrow F \cup \{g_1, \ldots, g_k\}$
        for each ordered pair of gates $(g_i, g_j)$ do
22:         while there is an $a$ s.t. $(g_i, a, q) \in \delta$
                  for $q$ outside Orbit($g_i$) and $(g_j, a, q) \notin \delta$ do
24:             Add $(g_j, a, q)$ to $\delta$
                while $(g_j, a, q') \in \delta$ for $q' \neq q$ do
26:                 Merge($q$,$q'$)
```

(Figure 3(b)). In the next recursive level, we call AHONEN *for every orbit of A. We only consider the non-trivial orbit* $\{2, 3, 4\}$ *here, with its orbit automaton* A_2 *in Figure 3(c). As there are no* A_2*-consistent symbols and* A_2 *has only one orbit, we recursively merge states 2 and 4. (Line 7 gives us a choice of which transition to take, but any choice would lead us to the merging of 2 and 4.) After the merge, a is* A_2*-consistent. We therefore call ForceOrbitproperty on the* $\{a\}$*-cut of* A_2 *as in Figure 3(e). Here, we discover that there are only two trivial orbits left, and the algorithm ends.*

It remains to return from the recursion and construct the resulting DFA of the algorithm. Plugging the DFA from Figure 3(d) into the DFA from Figure 3(b) results in Figure 3(f). Notice that this automaton is non-deterministic. Therefore, we have to merge states 3 and 5 in order to restore determinism. The final DFA obtained by the algorithm is in Figure 3(g). Notice that this is a non-minimal DFA defining the language $a(a + b)^*$.

Several additional remarks should be made about the original paper [1]: *(i)* it does not prove that the input DFA for Algorithm 3 is transformed into an automaton that can always be converted into a deterministic regular expression; *(ii)* it does not formally explain how states of the automaton should be merged; we have chosen the most reasonable definition; and *(iii)* it does not explain how the output DFA should be reconstructed when going back up in the recursion. Therefore, we had to make some assumptions. For example, we assume that, when re-combining orbits into a large automaton, we have a transition (q_1, a, q_2) if and only if there were subsets $q_1' \subseteq q_1$ and $q_2' \subseteq q_2$ such that the original automaton had a transition (q_1', a, q_2'). (See, for example, the transition $(\{2, 4\}, b, \{5\})$ in Figure 3(f), which is there because the original automaton in Figure 3(b) had a transition $(\{4\}, b, \{5\})$.)

With respect to remark *(i)*, we noticed in our experiments that Ahonen's algorithm sometimes indeed outputs a DFA that cannot be converted into an equivalent deterministic expression. If this happens, we reiterate Ahonen's algorithm to the thus far constructed DFA until the resulting DFA defines a deterministic language.

5.4 Ahonen's Algorithm followed by Grow

Ahonen's algorithm AHONEN-BKW relies on BKW to construct the corresponding deterministic expression. However, as we know, BKW generates very large expressions. We therefore consider the algorithm AHONEN-GROW which runs GROW on the DFA resulting from AHONEN.

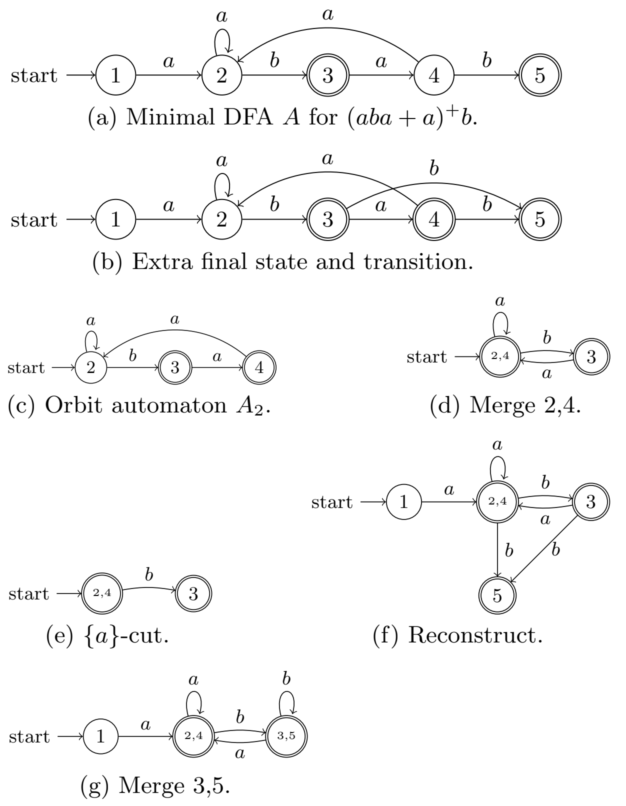

start
a
b
1
2
3
4
5
2,4
3,5
(a) Minimal DFA A for $(aba + a)^+b$.
(b) Extra final state and transition.
(c) Orbit automaton A_2.
(d) Merge 2,4.
(e) $\{a\}$-cut.
(f) Reconstruct.
(g) Merge 3,5.

Figure 3: Example run of the adapted Ahonen's algorithm.

5.5 Shrink

As a final approach, we present SHRINK. The latter algorithm operates on *state-labeled finite automata* instead of standard DFAs as we explain next. A state-labeled finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ is simply a finite automaton, which has the additional property that for each state q all transitions to q carry the same label. That is, there may not exist states q_1, q_2, and symbols a, b, with $(q_1, a, q), (q_2, b, q) \in \delta$, but $a \neq b$. If A is state-labeled we can associate a function lab : $Q \to \Sigma$ with A, such that $\text{lab}(q) = a$ iff a is the label of q, i.e., if there exists a q' such that $(q', a, q) \in \delta$. For instance, the automaton in Figure 3(a) is state-labeled and has $\text{lab}(2) = a$, $\text{lab}(3) = b$, $\text{lab}(4) = a$, and $\text{lab}(5) = b$; lab(1) is undefined as 1 does not have incoming transitions.

We note that from any finite automaton, we can easily construct an equivalent state-labeled automaton by duplicating states which have more than one symbol on their incoming transitions. In particular, from a minimal DFA, one can thus construct a minimal state-labeled DFA.

The philosophy behind SHRINK rather opposes the one behind GROW: it tries to reduce the number of states of the input automaton by merging pairs of states with the same label, until every state has a different label.The result of SHRINK is an array containing deterministic expressions for which the language proximity to the target language is maximal among the deterministic expressions of the same size.

SHRINK is presented in Algorithm 5. The call to MERGE(B, q_1, q_2) is similar to the one we explained in Section 5.3 and operates on the DFA B. It makes q_1 initial (resp., final) if q_2 was initial (resp., final). If this operation does not result in a DFA, we continue recursively. RANK(P,$\{B_1, \dots, B_j\}$) is a ranking procedure that selects the P "best" automata from the set $\{B_1, \dots, B_j\}$. Thereto, we say that an automaton B_i is *better* than B_j when $L(B_i)$ is deterministic but $L(B_j)$ is nondeterministic. Otherwise, if $L(B_i)$ and $L(B_j)$ are either both deterministic or both nondeterministic, B_i is better than B_j iff $\text{proximity}(B_i, A) > \text{proximity}(B_j, A)$. That is, we favour automata which define deterministic languages (as we are looking for deterministic languages), and when no distinction is made in this manner, we favour those automata which form the best approximation of the original language.

Algorithm 5 The SHRINK algorithm with pool size P.

```
Input: Minimal state-labeled DFA $A$, $P \in \mathbb{N}$
Output: Array of det. reg. exp. $s$ with $L(A) \subseteq L(s)$
    Pool ← $\{A\}$
    BestArray ← empty array of $|A| - |\Sigma|$ elements
    while Pool is not empty do
        $j \leftarrow 1$
        for each $B \in$ Pool do
            for each pair of states $q_1, q_2$ of $B$
                                   with $\text{lab}(q_1) = \text{lab}(q_2)$ do
                $B_j \leftarrow$ MERGE($B$,$q_1$,$q_2$)
                $j \leftarrow j + 1$
        Pool ← RANK($P$,$\{B_1, \dots, B_{j-1}\}$)
        for each $B_k \in$ Pool do
            $r_{k,1} \leftarrow$ KOA-TO-KORE($B_k$)
            $r_{k,2} \leftarrow$ GROW($B_k$)
        for each $\ell$, $|\Sigma| \leq \ell \leq |A|$ do
            BestArray[$\ell$] ← the deterministic regexp $r$ of size $\ell$
                     from BestArray[$\ell$] and all $r_{k,x}$ with
                            maximal value proximity($A, r$)
return BestArray
```

KOA-TO-KORE is an algorithm of [6] which transforms a state-labeled automaton A into a (possibly nondeterministic) expression r such that $L(A) \subseteq L(r)$. Further r contains one symbol for every labeled state in A. As we are only interested in deterministic expressions, we discard the result of KOA-TO-KORE when it is nondeterministic. However, if every state of A is labeled with a different symbol, then the resulting expression also contains every symbol only once, and hence is deterministic. As SHRINK will always generate automata which have this property, SHRINK is thus guaranteed to always output at least one deterministic approximation.

Notice that SHRINK always terminates: the automata in Pool become smaller in each iteration and when they have $|\Sigma|$ states left, no more merges can be performed.

6. EXPERIMENTS

In this section we validate our approach by means of an experimental analysis. All experiments were performed using a prototype implementation written in Java executed on a Pentium M 2 GHz with 1GB RAM. As the XML Schema specification forbids ambiguous content models, it is not possible to extract real-world expressions violating UPA from the Web. We therefore test our algorithms on a sizable and diverse set of generated regular expressions. To this end, we apply the synthetic regular expression generator used in [6] to generate 2100 nondeterministic regular expressions. From this set, 1200 define deterministic languages while the others do not. We utilize three parameters to obtain a versatile sample. The first parameter is the *size* of the expressions (number of occurrences of alphabet symbols) and ranges from 5 to 50. The second parameter is the *average number of occurrences* of alphabet symbols in the expression, denoted by κ. That is, when the size of r is n, then $\kappa(r) = n/|\Sigma(r)|$, where $\Sigma(r)$ is the set of different alphabet symbols occurring in r. For instance, when $r = a(a+b)^{+}acab$, then $\kappa(r) = 7/3 = 2.3$. In our sample, κ ranges from 1 to 5. At first glance, the maximum value of 5 for κ might seem small. However, the latter value must not be confused with the maximum number of occurrences of a single alphabet symbol, which in our sample ranges from 1 to 10. Finally, the third parameter measures how much the language of a generated expression overlaps with Σ^*, which is measured by proximity(r, Σ^*). The expressions are generated in such a way that the parameter covers the complete spectrum uniformly from 0 to 1.

6.1 Deciding determinism

As a sanity check, we first ran the algorithm BKW$^{\text{dec}}$ on the real world deterministic expressions obtained in the study [5] (which are all deterministic). On average they were decided to define a deterministic regular language within 35 milliseconds. This outcome is not very surprising as κ for each of these expressions is close to 1 (cfr. Proposition 4). We then ran the algorithm BKW$^{\text{dec}}$ on each of the 2100 expressions and were surprised that on average no more than 50 milliseconds were needed, even for the largest expressions of size 50. Upon examining these expressions more closely, we discovered that all of them have small corresponding *minimal* DFAs: on average 25 states or less. Apparently random regular expressions suffer much less from the theoretical worst case exponential size increase when translated into DFAs.

6.1.1 Discussion

Although the problem of deciding determinism is theoretically intractable (Theorem 3), in practice, there does not seem to be a problem so we can safely use BKW$^{\text{dec}}$ as a basic building block of Algorithm 6.

6.2 Constructing deterministic regular expressions

In this section, we compare the deterministic regular expressions generated by the three algorithms: BKW, BKW-OPT, and GROW. We point out that the comparison with BKW is not a fair one, as the latter was not defined with efficiency in mind.

Table 1 depicts the average sizes of the expressions generated by the three methods (again size refers to number of symbol occurrences), with the average running times in brackets. Input size refers to the size of the input regular expressions. Here, the pool-size and depth of GROW are 100 and 5, respectively. We note that for every expression individually the output of BKW is always larger than that of BKW-OPT and that GROW, when it succeeds, always gives the smallest expression. Due to the exponential nature of the BKW algorithm, both BKW and BKW-OPT can not be used for input expressions of size larger than 20.[4] For smaller

input size	BKW	BKW-OPT	GROW
5	9 (< 0.1)	7 (< 0.1)	3 (< 0.1)
10	216 (< 0.1)	95 (0.1)	6 (0.2)
15	1577 (0.2)	394 (0.6)	9 (0.6)
20	/	/	12 (1.5)
25-30	/	/	13 (4.0)
35-50	/	/	23 (19.6)

Table 1: Average output sizes and running times (in brackets, in seconds) of BKW, BKW-OPT and GROW on expressions of different input size.

size	(d:5,p:20)	(5,100)	(10,20)	(10,100)
5	89 (< 0.1)	89 (< 0.1)	89 (< 0.1)	89 (< 0.1)
10	66 (< 0.1)	68 (0.2)	68 (0.1)	70 (0.5)
15	43 (0.1)	46 (0.6)	44 (0.3)	47 (1.6)
20	31 (0.3)	33 (1.5)	31 (0.8)	33 (3.8)
25-30	21 (0.8)	21 (4.0)	21 (1.8)	21 (9.1)
35-50	7 (3.9)	8 (19.6)	7 (8.3)	8 (43.7)

Table 2: Success rates (%) and average running times (in brackets, in seconds) of GROW for different values of the depth (d) and pool-size (p) parameters.

input expressions, BKW-OPT is better than BKW, but still returns expressions which are in general too large to be easily interpreted. In strong contrast, when it succeeds, GROW produces very concise expressions, roughly the size of the input expression.

It remains to discuss the effectiveness of GROW. In Table 2, we give the success rates and the average running times for various sizes of input expressions and for several values for pool-size and depth. It is readily seen that the success rate of GROW is inversely proportional to the input size, starting at 90% for input size 5, but deteriorating to 20% for input size 25. Further, Table 2 also shows that increasing the pool-size or depth only has a minor impact on the success rate of GROW, but a bigger influence on its running time.

6.2.1 Discussion

GROW is the preferred method to run in a first try. When it gives a result it is always a concise one. Its success rate is inversely proportional to the size of the input expressions and quite reasonable for expressions up to size 20. Should GROW fail, it is not a real option to try BKW-OPT as on expressions of that size it never produces a reasonable result. In that case, the best option is to look for a concise approximation (as implemented in Algorithm 6).

6.3 Approximating deterministic regular expressions

We now compare the algorithms AHONEN-BKW, SHRINK, and AHONEN-GROW. Note that AHONEN-BKW and AHONEN-GROW return a single approximation, whereas SHRINK returns a set of expressions (with a tradeoff between size and proximity). To simplify the discussion, we take from the output of SHRINK the expression with the best proximity, disregarding the size of the expressions. This is justified as all expressions returned by SHRINK are concise by definition. In a practical scenario, however, the choice in tradeoff between proximity and conciseness can be left to the user.

[4]For expressions of size 20, BKW already returned expressions of size 560.000.

input size	AHONEN-BKW	AHONEN-GROW	SHRINK
5	0.73 (100%)	0.71 (75%)	0.75 (100%)
10	0.81 (100%)	0.79 (56%)	0.78 (100%)
15	0.84 (100%)	0.88 (40%)	0.79 (100%)
20	/	0.89 (18%)	0.76 (100%)
25-30	/	0.89 (8%)	0.71 (100%)
35-50	/	0.75 (4%)	0.68 (100%)

Table 3: Quality of approximations of AHONEN-BKW, AHONEN-GROW, and SHRINK (closer to one is better). Success rates in brackets.

input size	AHONEN-BKW	AHONEN-GROW	SHRINK
5	8 (100%)	3 (75%)	3 (100%)
10	28 (100%)	6 (56%)	6 (100%)
15	73 (100%)	8 (40%)	8 (100%)
20	/	11 (18%)	10 (100%)
25-30	/	11 (8%)	13 (100%)
35-50	/	14 (4%)	18 (100%)

Table 4: Average output sizes of AHONEN-BKW, AHONEN-GROW, and SHRINK. Success rates in brackets.

Table 3 then shows the average proximity(r, s) where r is the input expression and s is the expression produced by the algorithm. As the AHONEN-GROW algorithm is not guaranteed to produce an expression, the success rates are given in brackets. In contrast, AHONEN-BKW and SHRINK always return a deterministic expression. Further, as AHONEN-BKW uses BKW as a subroutine its use is restricted to input expressions of size 15.

We make several observations concerning Table 3. First, we see that the succes rate of AHONEN-GROW is inversely proportional to the size of the input expression. This is to be expected, as GROW is not very successful on large input expressions or automata. But, as AHONEN-BKW is also not suited for larger input expressions, only SHRINK produces results in this segment.

Concerning the quality of the approximations, we only compare AHONEN-BKW with SHRINK because AHONEN-GROW, when it succeeds, returns an expression equivalent to AHONEN-BKW which consequently possesses the same proximity. In Table 3, we observe that AHONEN-BKW returns on average slightly better approximations than SHRINK. Also in absolute values, AHONEN-BKW returns in roughly 2/3th of the cases the best approximation w.r.t. proximity, and SHRINK in the other 1/3th. We further observe that the quality of the approximations of SHRINK only slightly decreases but overall remains fairly good, with 0.68 for expressions of size 50.

Table 4 shows the average output sizes of the different algorithms. Here, we see the advantage of AHONEN-GROW over AHONEN-BKW. When AHONEN-GROW returns an expression it is much more concise than (though equivalent to) the output of AHONEN-BKW. Furthermore, it has a small chance on success for those sizes of expressions on which AHONEN-BKW is not feasible anymore. Also SHRINK can be seen to always return very concise expressions.

Finally, we consider running times. On the input sizes for which AHONEN-BKW is applicable, it runs in less than a second. AHONEN-GROW was executed with pool-size 100 and depth 5 for the GROW subroutine, and took less than a second for the small input sizes (5 to 15) and up to a half a minute for the largest (50). Finally, SHRINK was executed with pool-size 10, for input expressions of size 5 to 20, and pool-size 5 for bigger expressions, and took up to a few seconds for small input expressions, and a minute on average for the largest ones.

6.3.1 Discussion

As running times do not pose any restriction on the applicability of the proposed methods, the best option is to always try AHONEN-GROW and SHRINK, and AHONEN-BKW only for very small input expressions (up to size 5) and subsequently pick the best expression.

Algorithm 6 Supportive UPA Checker SUPAC.

```
Input: regular expression $r$
Output: deterministic reg. exp. $s$ with $L(r) \subseteq L(s)$
    if $r$ is deterministic then return $r$;
 2: else if $L(r)$ is deterministic then
      if GROW$(r)$ succeeds then return GROW$(r)$
 4:   else return best from BKW-OPT(r) and SHRINK$(r)$
    else return best from AHONEN-GROW(r) and SHRINK$(r)$
```

input	output
$c^*cac + b$	$c^+ac + b$
$(a?bc + d)^+d$	$((a?(bc)^+)^*d^+)^+$
$((cba + c)^*b)?$	$((c^+ba?)^*b?)?$
$(c^+cb + a + c)^*$	$(a^+ + c^+b?)^*$

Table 5: Sample output of SUPAC.

7. SUPAC: SUPPORTIVE UPA CHECKER

Based on the observations made in Section 6, we define our supportive UPA checker SUPAC as in Algorithm 6. We stress that the notion of 'best' expression can depend on both the conciseness and the proximity of the resulting regular expressions and is essentially left as a choice for the user. Note that, when GROW does not succeed, there is only a choice between a probably lengthy equivalent expressions generated by BKW-OPT or a concise approximation generated by SHRINK. In line 5, we could also make the distinction between expressions r of small size (≈ 5) and larger size (> 5). For small expressions, we then could also try AHONEN-BKW.

As an illustration, Table 5 lists a few small input expressions with the corresponding expression constructed by SUPAC. The first two expressions define deterministic languages, while the last two do not.

8. CONCLUSION

It is worth mentioning that none of the authors believe that the UPA constraint is a sensible one. Only a very limited and in practice negligible efficiency in parsing is gained at the expense of introducing a difficult notion which on top breaks the robustness of the larger class of regular expressions. Nevertheless, UPA is enforced by the XML Schema specification constituting a permanent burden for the designer. Therefore, the present paper proposes the algorithm SUPAC as a supportive UPA checker for handling nondeterministic regular expressions.

There are two immediate ways to improve SUPAC, but each of them requires an in depth study of its own. First, one can try to minimize obtained regular expressions. However, in strong contrast to their automaton counterparts, very little work has been done on minimizing of expression let alone deterministic ones. A possible starting point is a paper by Salomaa [29] which presents a sound and complete rewrite system for regular expressions. Some of these rules conserve determinism, but not all of them. Minimizing deterministic regular expressions is easily seen to be in NP, but it is not clear whether it is also hard. A second possibility for improvement is to apply combinatorial optimization techniques to replace the brute force flavored subroutines in GROW and SHRINK. An immediate difficulty, however, is how to choose between the different possibilities for local changes applied to the automata. It is unclear how to score such local changes and assess their ability to generate a concise deterministic regular expression. In fact, one of the results of this paper, is that sophisticated methods like BKW (and its optimized version BKW-OPT) and its variant with repair rules AHONEN-BKW do not produce reasonable results.

In this paper we ignored numerical occurrence indicators as in $c^{[1,2]}(a^{[4,5]} + b)^*$. Denote by $\mathrm{RE}^{\#}$ the set of regular expressions with numerical occurrence indicators. Kilpeläinen and Tuhkanen [20] introduce an algorithm which decides whether $\mathrm{RE}^{\#}$-expressions are deterministic, but it is not known how to decide whether a regular language can be represented by a deterministic $\mathrm{RE}^{\#}$. Obtaining such a result would be a major step forward.

9. REFERENCES

[1] H. Ahonen. Disambiguation of SGML content models. In *Workshop on Principles of Document Processing (PODP)*, p. 27–37, 1996.

[2] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: gathering statistics from an XML sample. *World Wide Web*, 8(4):413–438, 2005.

[3] M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2), 2007.

[4] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[5] G.J.Bex, F.Neven, J.Van den Bussche. DTDs versus XML Schema: a practical study. *Workshop on the Web and Databases (WebDB)*, p. 79-84, 2004.

[6] G.J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *World Wide Web Conference (WWW)*, p. 825–834, 2008.

[7] G.J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *Very Large Data Bases (VLDB)*, p. 115–126, 2006.

[8] G.J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *Very Large Data Bases (VLDB)*, p. 998–1009, 2007.

[9] G.J. Bex, F. Neven, and S. Vansummeren. SchemaScope: a system for inferring and cleaning XML schemas. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, p. 1259–1262, 2008.

[10] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142:182–206, 1998.

[11] D. Che, K. Aberer, and M. T. Özsu. Query optimization in XML structured-document databases. *VLDB Journal*, 15(3):263–289, 2006.

[12] C. Chitic and D. Rosu. On validation of XML streams using finite state machines. In *Workshop on the Web and Databases (WebDB)*, p. 85–90, 2004.

[13] J. Freire, F. Du, S. Amer-Yahia. ShreX: Managing XML Documents in Relational Databases. In *Very Large Data Bases (VLDB)*, p. 1297–1300, 2004.

[14] J. Flum and M. Grohe. *Parametrized Complexity Theory*. Springer, 2006.

[15] J. Freire, J.R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX:making XML count. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, p. 181–191, 2002.

[16] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, p. 325–336, 2008.

[17] G. Ghelli, D. Colazzo, C. Sartiani. Efficient inclusion for a class of xml types with interleaving and counting. In *Database Programming Languages (DBPL)*, p. 231–245, 2007.

[18] G. Ghelli, D. Colazzo, and C. Sartiani. Linear time membership in a class of regular expressions with interleaving and counting. In *Conference on Information and Knowledge Management (CIKM)*, p. 389–398, 2008.

[19] H. Gruber and J. Johannsen. Optimal lower bounds on regular expression size using communication complexity. In *Foundations of Software Science and Computation Structures*, p. 273–286, 2008.

[20] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.

[21] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB Journal*, 16(3):317–342, 2007.

[22] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Very Large Data Bases (VLDB)*, p. 228–239, 2004.

[23] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Very Large Data Bases (VLDB)*, p. 241–250, 2001.

[24] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science (MFCS)*, p. 889–900, 2004.

[25] W. Martens, F. Neven, T. Schwentick, and G.J. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[26] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *World Wide Web Conference (WWW)*, p. 500–510, Budapest, Hungary, 2003.

[27] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction,

DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[28] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[29] A.Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*,13:158–169, 1966.

[30] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema part 1: Structures.* World Wide Web Consortium (W3C), May 2001.

[31] E. van der Vlist. *XML Schema.* O'Reilly, 2002.

[32] P. van Emde Boas. The convenience of tilings. In *Complexity, Logic and Recursion Theory*, p. 331–363.

APPENDIX

A. MISSING PROOFS

Proof of Theorem 3: *Given a regular expression r, the problem of deciding whether $L(r)$ is deterministic is* PSPACE*-hard.*

PROOF. A *tiling instance* is a tuple $T = (X, H, V, \bar{b}, \bar{t}, n)$ where X is a finite set of tiles, $H, V \subseteq X \times X$ are the horizontal and vertical constraints, and $\bar{b}, \bar{t}$ are n-tuples of tiles ($\bar{b}$ and $\bar{t}$ stand for *bottom row* and *top row*, respectively).

A *correct corridor tiling for* T is a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \to X$ for some $m \in \mathbb{N}$ such that the following constraints are satisfied:

- the bottom row is $\bar{b}$: $\bar{b} = (\lambda(1, 1), \ldots, \lambda(1, n))$;
- the top row is $\bar{t}$: $\bar{t} = (\lambda(m, 1), \ldots, \lambda(m, n))$;
- all vertical constraints are satisfied: $\forall 1 \leq i < m$, $\forall 1 \leq j \leq n$, $(\lambda(i, j), \lambda(i + 1, j)) \in V$; and,
- all horizontal constraints are satisfied: $\forall 1 \leq i \leq m$, $\forall 1 \leq j < n$, $(\lambda(i, j), \lambda(i, j + 1)) \in H$.

The CORRIDOR TILING problem asks, given a tiling instance, whether there exists a correct corridor tiling. The latter problem is PSPACE-complete [32].

We reduce from CORRIDOR TILING. However, we restrict ourselves to those tiling instances for which there exists at most one correct corridor tiling. Notice that we can assume this without loss of generality: From the master reduction from Turing Machine acceptance to CORRIDOR TILING in [32], it follows that the number of correct tilings of the constructed tiling system is precisely the number of accepting runs of the Turing Machine on its input word. As the acceptance problem for polynomial space bounded Turing Machines is already PSPACE-complete for *deterministic* machines, we can assume w.l.o.g. that the input instance of CORRIDOR TILING has at most one correct corridor tiling.

Now, let T be a tiling instance for which there exists at most one correct tiling. We construct a regular expression r, such that $L(r)$ is deterministic iff there does not exist a corridor tiling for T. Before giving the actual definition of r, we give the language it will define and show this is indeed deterministic iff CORRIDOR TILING for T is false. We encode corridor tilings by a string in which the different rows are separated by the symbol \$, that is, by strings of the form

$$\$R_1\$R_2\$\cdots\$R_m\$$$

in which each R_i represents a row and is therefore in X^n. Moreover, R_1 is the bottom row and R_n is the top row.

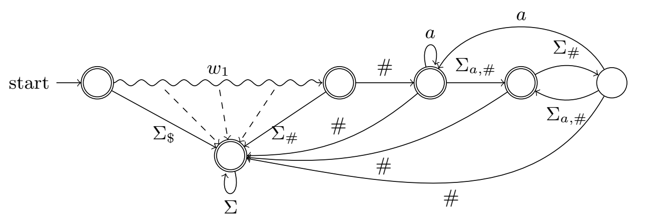


Figure 4: DFA for $L(r)$ in the proof of Theorem 3.

Then, let $\Sigma = X \uplus \{a, \$, \#\}$ and for a symbol $b \in \Sigma$, denote by $\Sigma_b = \Sigma \setminus \{b\}$. Then, $L(r) = \Sigma^* \setminus \{w_1\#w_2 \mid w_1$ encodes a valid tiling for T and $w_2 \in \Sigma_\#^*\Sigma_{a,\#}\Sigma_\#\}$. First, if there does not exist a valid tiling for T, then $L(r) = \Sigma^*$ and thus $L(r)$ is deterministic. Conversely, if there does exist a valid corridor tiling for T, then by our assumption, there exists exactly one. A DFA for $L(r)$ is graphically illustrated in Figure 4. Notice that this DFA is the minimal DFA iff w_1 exists. By applying the algorithm of Brüggemann-Klein and Wood (Algorithm 2), it is easily seen that $L(r)$ is not deterministic. Indeed, Algorithm 2 gets immediately stuck in line 17, where it sees that the gates in the orbit consisting of the three rightmost states in Figure 4 are not all final states. Hence, this minimal DFA does not satisfy the orbit property.

Our regular expression r now consists of the disjunction of the following regular expressions:[5]

- $\Sigma^*\#\Sigma^*\#\Sigma^* + \Sigma_\#^*$: This expression detects strings that don't have exactly one occurrence of $\#$.
- $\Sigma^*\#\Sigma?$: This expression detects strings that have $\#$ as last or second to last symbol.
- $\Sigma_\#^*\Sigma^*a\Sigma$: This expression detects strings that have a as second to last symbol.
- $\Sigma^*a\Sigma^*\#\Sigma^*$: This expression detects strings that have an a before the $\#$-sign.
- $\Sigma_\$\Sigma^* + \Sigma^*\Sigma_\$\#\Sigma^*$: This expression detects strings that don't have a \$-sign as the first or last element of their encoding.
- $\Sigma^*\$\Sigma_\$^{[0,n-1]}\$\Sigma^*\#\Sigma^* + \Sigma^*\$\Sigma_\$^{[n+1,n+1]}\Sigma_\$^*\$\Sigma^*\#\Sigma^*$: This expression detects all string in which a row in the tiling encoding is too short or too long.
- $\Sigma^*x_1x_2\Sigma^*\#\Sigma^*$, for every $x_1, x_2 \in X, (x_1, x_2) \notin H$: These expressions detect all violations of horizontal constraints in the tiling encoding.
- $\Sigma^*x_1\Sigma^nx_2\Sigma^*\#\Sigma^*$, for every $x_1, x_2 \in X, (x_1, x_2) \notin V$: These expressions detect all violations of vertical constraints in the tiling encoding.
- $\Sigma^{i+1}\Sigma_{\bar{b}_i}\Sigma^*\#\Sigma^*$ for every $1 \leq i < n$: These expressions detect all tilings which do not have $\bar{b}$ as the bottom row in the tiling encoding.
- $\Sigma^*\Sigma_{\bar{t}_i}\Sigma^{n-i}\#\Sigma^*$ for every $1 \leq i < n$: These expressions detect all tilings which do not have $\bar{t}$ as the top row in the tiling encoding.

Finally, it is easily verified that $L(r)$ is defined correctly. □

LEMMA 9. *For every deterministic language L and string $w \in L$, the language $L \setminus \{w\}$ is also deterministic.*

[5] Notice that r itself doesn't have to be deterministic.

PROOF. By $|u|$ we denote the length of string u. We define the prefix-language $L^{\leq |w|} = \{u \in L \mid |u| < |w|\} \cup \{u \in \Sigma^* \mid |u| = |w|, \exists v.uv \in L\}$. As $L^{\leq k}$ is a finite language, one can easily construct a deterministic regular expression for it consisting of nested disjunctions. For instance, the set $L^{\leq |w|} = \{aab, ab, baa, bba\}$ can be defined by $aa(b+\varepsilon) + b(aa+ba)$. Denote the resulting expression by r. Further, we note that deterministic regular languages are closed under *derivatives* [10]. Specifically, for a string u, the u-derivative of a regular expression s is a regular expression s' that defines the set $\{v \mid uv \in L(s)\}$. For every string $u \in L^{\leq |w|}$ with $|u| = |w|$ and $u \neq w$, let r_u be a deterministic expression defining the u-derivative of L. Notice that the u-derivative can be ε. Now for $u = w$, let r_w define the w-derivative of L minus ε. It is shown in [10] (Theorem D.4) that, for every deterministic regular language L', $L' - \{\varepsilon\}$ is also deterministic. Hence, r_w can also be written as a deterministic regular expression. Now, the expression defining $L \setminus \{w\}$ is obtained by adding, for each $u \in L^{\leq |w|}$ with $|u| = |w|$, r_u after the last symbol of u in r. □

LEMMA 10. *For every deterministic language L and string $w \notin L$, the language $L \cup \{w\}$ is also deterministic.*

PROOF. Analogous to the proof of Lemma 9 □

Algorithm 7 The original FORCEORBITPROPERTY.

```
   ForceOrbitProperty($A = (Q, \Sigma, \delta, q_0, F)$)
 2: for each orbit $C$ of $A$ do
      Let $g_1, \ldots, g_k$ be the gates of $K$
 4:   if there exists a gate $g_i \in F$ then
        $F \leftarrow F \cup \{g_1, \ldots, g_k\}$
 6:   for each ordered pair of gates $(g_i, g_j)$ do
        if there is an $a$ s.t. $(g_i, a, q) \in \delta$
 8:          for $q$ outside Orbit($g_i$) and $(g_j, a, q) \notin \delta$ then
          if $(q_j, a, q') \in \delta$ for $q' \neq q$ then
10:         Add $(g_j, a, q)$ to $\delta$
            Merge $q$ and $q'$
```

B. AHONEN'S ORIGINAL ALGORITHM

Ahonen's original algorithm [1] is essentially the same as the one we already presented, with a slightly different FORCEORBITPROPERTY (see Algorithm 7). We did not succeed in recovering the actual implementation of Ahonen. As the algorithm presented by Ahonen is incorrect, we had to mend it in order to make a fair comparison. In particular, we observed the following: (a) The if-test on l.9 should not be there. Otherwise, the output will certainly not always be an automaton that fulfills the orbit property. (b) The if-test on l.8 should, in our opinion, be some kind of for-loop. Currently, the algorithm does not necessarily choose the same a for each pair of gates (g_i, g_j). We do believe, however, that these differences are merely typos and that Ahonen actually intended to present this new version.