Schema-Based Automata Determinization

Joachim Niehren
Inria, France Université de Lille
joachim.niehren@inria.fr

Momar Sakho
Inria, France Université de Lille
momar.sakho@inria.fr

Antonio Al Serhali
Inria, France Université de Lille
antonio.al-serhali@inria.fr

We propose an algorithm for schema-based determinization of finite automata on words and of stepwise hedge automata on nested words. The idea is to integrate schema-based cleaning directly into automata determinization. We prove the correctness of our new algorithm and show that it is always more efficient than standard determinization followed by schema-based cleaning. Our implementation permits to obtain a small deterministic automaton for an example of an XPath query, where standard determinization yields a huge stepwise hedge automaton for which schema-based cleaning runs out of memory.

1 Introduction

Nested words are words enhanced with well-nested parenthesis. They generalize over trees, unranked trees, and sequences of thereof that are also called hedges or forests. Nested words provide a formal way to represent semi-structured textual documents of XML and JSON format.

Regular queries for nested words can be defined by finite state automata. We will use stepwise hedge automata (SHAs) for this purpose [19], which combine finite state automata for words and trees in a natural manner. SHAs refine previous notions of hedge automata from the sixties [24, 9] in order to obtain a decent notion of left-to-right and bottom-up determinism. They extend on stepwise tree automata [8], so that they can not only be applied to unranked trees but also to hedges. Any SHA defines a forest algebra [5] based on its transition relation. Furthermore, SHAs can always be determinized and have the same expressiveness as (deterministic) nested word automaton (NWA) [16, 6, 2, 21]. Note, however, that SHAs do not provide any form of top-down determinism in contrast to NWA*s*.

Efficient compilers from regular XPATH queries to SHAs exist [19], possibly using NWA*s* as intermediates [4, 17, 10]. Our main motivation is to determinize the SHAs of regular XPATH queries since deterministic automata are crucial for various algorithmic querying tasks. In particular, determinism reduce the complexity of universality or inclusion checking from EXP-completeness to P-time, both for the classes of deterministic SHAs or NWA*s*. In turn, universality checking is relevant for the earliest query answering of XPath queries on XML streams [14]. Furthermore, determinism is needed for efficient in-memory answer enumeration of regular queries [22].

Automata determinization may take exponential time in the worst case, so it may not always be feasible in practice. For SHAs compiled from the XPATH queries of the XPathMark benchmark [12], however, it was shown to be unproblematic. This changes for the XPATH benchmark collected by Lick and Schmitz [15]: for 37% of its regular XPATH queries, SHA determinization does require more than 100 seconds, in which case it produces huge deterministic automata [1]. An example is:

```
(QN7)     /a/b//(* | @* | comment() | text())
```

This XPath query selects all nodes of an XML document that are descendants of a *b*-element below an *a*-element at the root. The nodes may have any XML type: element, attribute, comment, or text. The

D. Della Monica and P. Ganty (Eds.): 13th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 22)
EPTCS 370, 2022, pp. 49–65, doi:10.4204/EPTCS.370.4

© Niehren, Sakho, and Al Serhali
This work is licensed under the Creative Commons Attribution License.

nondeterministic SHA for QN7 has 145 states and an overall size of 348. Its determinization however leads to an automaton with 10.005 states and an overall size of 1.634.122.

A kick-off question is how to reduce the size of deterministic automata. One approach beside of minimization is to apply schema-based cleaning [19], where the schema of a query defines to which nested words the query can be applied. Schemas are always given by deterministic automata while the automata for queries may be nondeterministic. The idea of schema-based automaton cleaning is to keep only those states and transition rules of the automaton, that are needed to recognize some nested word satisfying the schema. The needed states and rules can be found by building the product of automata for the query and the schema. For XPATH queries selecting nodes, we have the schema one^x that states that a single node is selected for a fixed variable x by any answer of the query. The second schema expresses which nested words satisfy the XML data model. With the intersection of these two schemas, the schema-based cleaning of the deterministic SHA for QN7 indeed has only 74 states and 203 rules. When applying SHA minimization afterwards, the size of the automaton goes down to 27 states and 71 transition rules. However, our implementation of schema-based cleaning, runs out of memory for larger automata with say more than 1000 states. Therefore, we cannot compute the schema-based cleaning from the deterministic SHA obtained from QN7. Neither can we minimize it with our implementation of deterministic SHA minimization. The question of how to produce small deterministic automaton for queries as simple as QN7 thus needs a different answer.

Given the relevance of schemas, one naive approach could be to determinize the product of the automata for the query and schema. This may look questionable at first sight, given that the schema-product may be bigger than the original automaton, so why could it make determinization more efficient? But in the case of QN7, the determinization of the schema-product yields a deterministic automata with only 92 states and 325 transition rules, and can be computed efficiently. This observation is very promising, motivating three general questions:

1. Why are schemas so important for automata determinization?
2. Can this be established by some complexity result?
3. Is there a way to compute the schema-based cleaning of the determinization of an SHA more efficiently than be schema-based cleaning followed by determinization?

Our main result is a novel algorithm for schema-based determinization of NFAs and SHAs, that integrates schema-based cleaning directly into the usual determinization algorithm. This algorithm answers question 3 positively. Its idea is to keep only those subsets of states of the automaton during the determinization, that can be aligned to some state of the schema. In our Theorem 2, we prove that schema-based determinization always produces the same deterministic automaton than schema-free determinization followed by schema-based cleaning. By schema-based determinization we could compute the schema-based cleaning of the determinization of QN7 in less than three seconds. In contrast, the schema-based cleaning of the determinization does not terminate after a few hours. In the general case, the worst case complexity of schema-based determinization is lower than schema-less determinization followed by schema-based cleaning.

We also provide a more precise complexity upper bound in Proposition 13. Given an nondeterministic SHA A let $det(A)$ be its determinization, and given a deterministic SHA S for the schema, let $A \times S$ the accessible part of the schema-product, and $scl_S(A)$ the schema-based cleaning of S with respect to schema S. We show that the upper bound for the maximal computation time of $scl_S(det(A))$ depends quadratically on the number of states of $S \times det(A)$, which is often way smaller than for $det(A)$ since S is deterministic. This complexity result shows why the schema is so relevant for determinization

(questions 1 and 2), and why computing the schema-based determinization is often more efficient than determinization followed by schema-based cleaning (question 3).

To see that $S \times det(A)$ is often way smaller than $det(A)$ for deterministic S we first note that $det(A \times S) = det(A) \times S$ since S is deterministic.[1] So for the many states $Q = \{q_1 \dots q_n\}$ of $det(A)$ there may not exist any state s of S such that $(Q,s) \in det(A) \times S$, because this requires all states q_i can be aligned to s, i.e. that (q_i,s) in $A \times S$ for all $1 \leq i \leq n$. Furthermore, $\det(A) \times S$ is equal to $det_S(A) \times S$, so that $det(A \times S) = det_S(A) \times S$. Hence any size bound for the schema-based determinization $det_S(A)$ implies a size bound for the determinization of the schema-product. Also, in our experiments $det_S(A) \times S$ is almost by a factor of 2 bigger than $det_S(A)$. So the size of the determinization of the schema-product is closely tied to the size of the schema-based determinization.

We also present a experimental evaluation of our implementation of schema-based determinization of SHAs. We consider a scalable family of SHAs obtained from a scalable family of XPATH queries. Our experiments confirm the very large improvement implied by the usage of schemas for determinization. For this, we implemented the algorithm for schema-based SHA determinization in Scala. Furthermore, we applied the XSLT compiler from regular forward XPATH queries to SHAs from [19], as well as the datalog implementations of SHA minimization and schema-based cleaning from there.

A large scale experiment on practical XPATH queries was provided in follow-up work [1] where schema-based algorithms were applied to the regular XPATH queries collected by Lick and Schmitz [15] from real word XQuery and XSLT programs. Small deterministic SHAs could be obtained by schema-based determinization for all regular XPATH queries in this corpus. In contrast, standard determinization in 37% of the cases fails with a timeout of 100 seconds. Without this timeout, determinization either runs out of memory or produces very large automata.

Outline. We start with related work on automata for nested words, determinization for XPATH queries (Section 2). In Section 3, we recall the definition NFAs and discuss how to use them as schemas and queries on words. In Section 4, we recall schema-based cleaning for NFAs. In Section 5, we contribute our schema-based determinization algorithm in the case of NFAs and show its correctness. In Section 6, we recall the notion of SHAs for defining languages of nested words. In Section 7, we lift schema-based determinization to SHAs. Full proofs can be found in the Appendix of the long version [20].

2 Related Work

We focus on automata for nested words, even though our results are new for NFAs too.

Nested word automata. As recalled in the survey of Okhotin and Salomaa [21], Alur's et al. [2] NWA*s* were first introduced in the eighties under the name of input driven automata by Mehlhorn [16], and then reinvented several times under different names. In particular, they were called visibly pushdown automata [3], pushdown forest automata [18], and streaming tree automata [13]. The determinization algorithm for NWA*s* was first invented in the eighties by von Braunmühl and Verbeek in the journal version of [6] and then rediscovered various times later on too.

Determinization algorithms. The usual determinization algorithms for NFAs relies on the well-known subset construction. The determinization algorithms of bottom-up tree automata and SHAs are straightforward extensions thereof. The determinization algorithm for NWA*s*, in contrast, is more complicated, since having to deal with pushdowns. Subsets of *pairs* of states are to be considered there and not

[1] If $\{(q_1,s_1) \dots (q_n,s_n)\} \in det(A \times S)$ then there exists a tree that can go into all states $q_1 \dots q_n$ with A and into all states $s_1, \dots s_n$ with S. Since S is deterministic, we have $s_1 = \dots s_n$. So there exists a tree going into $\{q_1, \dots, q_n\}$ with $det(A)$ and also into all s_i. So $(\{q_1, \dots, q_n\}, s_i)$ is a state of $det(A) \times S$.

$$\frac{I^A \neq \emptyset}{I^A \in I^{det(A)} \qquad I^A \in \mathscr{Q}^{\det(A)}} \qquad \frac{Q \in \mathscr{Q}^{\det(A)} \qquad Q \cap F^A \neq \emptyset}{Q \in F^{\det(A)}}$$

$$\frac{Q \in \mathscr{Q}^{\det(A)} \qquad Q' = \{q' \in \mathscr{Q}^A \mid q \xrightarrow{a} q' \in \Delta^A,\ q \in Q\} \neq \emptyset}{Q \xrightarrow{a} Q' \in \Delta^{det(A)} \qquad Q' \in \mathscr{Q}^{\det(A)}}$$

$$\det(A) = (\Sigma, \mathscr{Q}^{\det(A)}, \Delta^{\det(A)}, I^{\det(A)}, F^{\det(A)})$$

Figure 1: The accessible determinization $\det(A)$ of NFA A.

only subsets of states as with the usual automata determinization algorithm. We also notice that general pushdown automata with nonvisible stacks can even not always be determinized.

Application to XPATH. Debarbieux et al. [10] noticed that the determinization algorithm for NWA*s* often behaves badly when applied to NWA*s* obtained from XPath queries as simple as $//a/b$. Niehren and Sakho [19] observed more recently that the situation is different for the determinization of SHAs: It works out nicely for the SHA of $//a/b$ and also for all other SHAs obtained by compilation from forward navigational XPath queries in the XPathMark benchmark [12]. Even more surprisingly, the same good behavior could be observed for the determinization algorithm of NWA when restricted to NWA*s* with the weak-single entry property.

Weak single-entry NWA*s* versus SHAs. The weak-single entry property implies that an NFA cannot memoize anything in its state when moving top-down. So it can only pass information left-to-right and and bottom-up, similarly to an SHA. This property failed for the NWA*s* considered by Debarbieux et al. and the determinization of their NWA*s* thus required top-down determinization. This quickly led to the size explosion described above. One the other hand side, the weak single-entry property can always be established in quadratic time by compiling NWA*s* to SHAs forth and back. Or else, one can avoid top-down determinization all over by directly working with SHAs as we do here.

3 Finite Automata on Words, Schemas, and Queries

In this section, we discuss hwo to use NFAs for defining schemas and queries on words.

Let $\mathbb{N}$ be the set of natural numbers including 0. The set of words over a finite alphabet Σ is $\Sigma^* = \bigcup_{n\in\mathbb{N}}^{\infty} \Sigma^n$. A word $(a_1,\ldots,a_n) \in \Sigma^n$ is written as $a_1 \ldots a_n$. We denote by ε the empty word, i.e., the unique element of Σ^0 and by $w_1 \cdot w_2 \in \Sigma^*$ the concatenation of two words $w_1, w_2 \in \Sigma^*$. For example, if $\Sigma = \{a,b\}$ then $aa \cdot bb = aabb = a \cdot a \cdot b \cdot b$.

Definition 1. *A* NFA *is a tuple $A = (\Sigma, \mathscr{Q}, \Delta, I, F)$ such that $\mathscr{Q}$ is a finite set of states, the alphabet Σ is a finite set, $I, F \subseteq \mathscr{Q}$ are subsets of initial and final states, and $\Delta \subseteq \mathscr{Q} \times \Sigma \times \mathscr{Q}$ is the set of transition rules.*

The size of a NFA is $|A| = |\mathscr{Q}| + |\Delta|$. A transition rule $(q,a,q') \in \Delta$ is denoted by $q \xrightarrow{a} q' \in \Delta$. We define transitions $q \xrightarrow{w} q'$ wrt Δ for arbitrary words $w \in \Sigma^*$ by the following inference rules:

$$\frac{q \in \mathscr{Q}}{q \xrightarrow{\varepsilon} q \text{ wrt } \Delta} \qquad \frac{q \xrightarrow{a} q' \in \Delta}{q \xrightarrow{a} q' \text{ wrt } \Delta} \qquad \frac{q_0 \xrightarrow{w_1} q_1 \text{ wrt } \Delta \qquad q_1 \xrightarrow{w_2} q_2 \text{ wrt } \Delta}{q_0 \xrightarrow{w_1 \cdot w_2} q_2 \text{ wrt } \Delta}$$

The language of words recognized by a NFA then is $\mathscr{L}(A) = \{w \in \Sigma^* \mid q \xrightarrow{w} q' \text{ wrt } \Delta,\ q \in I,\ q' \in F\}$.

```
1  fun det(A) =
2    let Store = hashset.new(∅) and Agenda = list.new() and Rules = hashset.new(∅)
3    if init^A ≠ ∅ then Agenda.add(init^A)
4    while Agenda.notEmpty() do
5      let Q = Agenda.pop()
6      let h be an empty hash table with keys from Σ.
7          // the values will be nonempty hash subsets of 2^A
8      for q -a-> q' ∈ Δ^A such that q ∈ Q do
9          if h.get(a) = undef then h.add(a,hashset.new(∅))
10         (h.get(a)).add(q')
11     for (a,Q') in h.tolist() do Rules.add(Q -a-> Q')
12         if not Store.member(Q') then Store.add(Q') Agenda.push(Q')
13   let init^det(A) = {Q | Q ∈ Store, Q ∩ init^A ≠ ∅} and F^det(A) = {Q | Q ∈ Store, Q ∩ F^A ≠ ∅}
14   return (Σ, Store.toSet(), Rules.toSet(), init^det(A), F^det(A))
```

Figure 2: A program computing the accessible determinization of an NFA A from Figure 1.

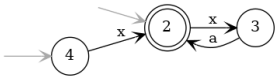


Figure 3: The NFA A_0 for the regular expression $(x+\varepsilon).(x.a)^*$

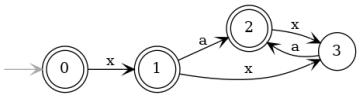


Figure 4: The accessible determinization $\det(A_0)$ up to the renaming of states $[\{2,4\}/0,\{2,3\}/1,\{2\}/2,\{3\}/3]$.

A NFA A is called *deterministic* or equivalently a DFA, if it has at most one initial state, and for every pair $(q,a) \in \mathcal{Q} \times \Sigma$ there is at most one state $q' \in \mathcal{Q}^A$ such that $q \xrightarrow{a} q' \in \Delta^A$. Any NFA A can be converted into a DFA that recognizes the same language by the usual subset construction. The accessible determinization $\det(A)$ of $A = (\Sigma, \mathcal{Q}^A, \Delta^A, I^A, F^A)$ is defined by the inference rules in Figure 1. It works like the usual subset construction, except that only accessible subsets are created. It is well known that $\mathcal{L}(A) = \mathcal{L}(\det(A))$. Since only accessible subsets of states are added, we have $\mathcal{Q}^{\det(A)} \subseteq 2^{\mathcal{Q}^A}$. Therefore, the accessible determinization may even reduce the size of the automaton and often avoid the exponential worst case where $\mathcal{Q}^{\det(A)} = 2^{\mathcal{Q}^A}$.

Proposition 2 (Folklore)**.** *The accessible determinization* $\det(A)$ *of a* NFA *A can be computed in expected amortized time* $O(|\mathcal{Q}^{\det(A)}|\ |\Delta^A| + |A|)$.

Proof sketch. The algorithm for accessible determinization with this complexity is somehow folklore. We sketch it nevertheless, since we need to refined it for schema-based determinization later on. A set of inference rules for accessible determinization is given in Figure 1, and an algorithm computing the fixed point of these inference rules is presented in Figure 2. It uses dynamic perfect hashing [11] for implementing hash sets, so that set inserting and membership can be done in randomized amortized time $O(1)$. The algorithm has a hash set *Store* to save all discovered states $\mathcal{Q}^{det(A)}$ and a hash set *Rules* to collect all transition rules. Furthermore, it has a stack *Agenda* to process all new states $Q \in \mathcal{Q}^{det(A)}$. □

As a running example, we consider the NFA A_0 for the regular expression $(x+\varepsilon).(x.a)^*$ that is drawn as a labeled digraph in Figure 3: the nodes of the graph are the states and the labeled edges represent the transitions rules. The initial states are indicated by an ingoing arrow and the final state are doubly circled. The graph of the DFA $\det(A_0)$ obtained by accessible determinization is shown in Figure 4. It is

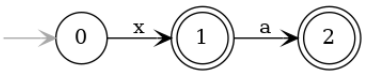


Figure 5: The schema-based cleaning of $\det(A_0)$ with schema $\mathit{words\text{-}one}^x_\Sigma$.

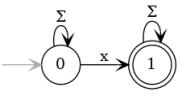
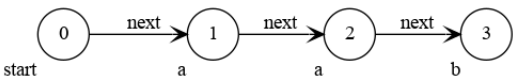


Figure 6: Schema $\mathit{words\text{-}one}^x_\Sigma$ with alphabet $\Sigma \uplus \{x\}$.

given up to a renaming of the states that is given in the caption. Note that only 4 out of the $2^3 = 8$ subsets are accessible, so the size increases only by a single state and two transitions rules in this example.

A *regular schema* over Σ is a DFA with the alphabet Σ. We next show how to use automata to define regular queries on words. For this, any word is seen as a labeled digraph. The labeled digraph of the word *aab*, for instance, is drawn to the right. The set of nodes of the graph is the set of positions of the word $\mathit{pos}(w) = \{0,\ldots,n\}$ where n is the length of w. Position 0 is labeled by *start*, while all other positions are labeled by a single letter in Σ. A monadic query function on words with alphabet Σ is a total function $\mathbf{Q}$ that maps some words $w \in \Sigma^*$ to a subset of position $\mathbf{Q}(w) \subseteq \mathit{pos}(w)$. We say that a position $\pi \in \mathit{pos}(w)$ is selected by $\mathbf{Q}$ if $w \in \mathit{dom}(\mathbf{Q})$ and $\pi \in \mathbf{Q}(w)$.



Let us fix a single variable x. Given a position π of a word $w \in \Sigma^*$ let $w * [\pi/x]$ be the word obtained from w by inserting x after position π. We note that all words of the form $w * [\pi/x]$ contain a single occurrence of x. Such words are also called V-structures where $V = \{x\}$ (see e.g [23]).

The set of all V-structures can be defined by the schema $\mathit{words\text{-}one}^x_\Sigma$ over $\Sigma \uplus \{x\}$ in Figure 6. It is natural to identify any total monadic query function $\mathbf{Q}$ with the language of V-structures $L_\mathbf{Q} = \{w * [\pi/x] \mid w \in \Sigma^*, \pi \in \mathbf{Q}(w)\}$. This view permits us to define a subclass of total monadic query functions by automata. A *(monadic) query automaton over* Σ is a NFA A with alphabet $\Sigma \uplus \{x\}$. It defines the unique total monadic query function $\mathbf{Q}$ such that $L_\mathbf{Q} = \mathscr{L}(A) \cap \mathscr{L}(\mathit{words\text{-}one}^x_\Sigma)$. A position π of a word $w \in \Sigma^*$ is thus selected by the query $\mathbf{Q}$ on w if and only if the V-structure $w * [\pi/x]$ is recognized by A, i.e.:

$$\pi \in \mathbf{Q}(w) \Leftrightarrow w * [\pi/x] \in \mathscr{L}(A)$$

A query function is called regular if it can be defined by some NFA. It is well-known from the work of Büchi in the sixties [7] that the same class of regular query functions can be defined equivalently by monadic second-order logic.

We note that only the words satisfying the schema $\mathit{words\text{-}one}^x_\Sigma$ (the V-structures) are relevant for the query function $\mathbf{Q}$ of a query automaton A. The query automaton A_0 in Figure 3 for instance, defines the query function that selects the start position of the words ε and a and no other positions elsewhere. This is since the subset of V-structures recognized by A_0 is $x + x.a$. Note that the words ε and xxa do also belong to $\mathscr{L}(A_0)$, but are not V-structures, and thus are irrelevant for the query function $\mathbf{Q}$.

4 Schema-Based Cleaning

Schema-based cleaning was introduced only recently [19] in order to reduce the size of automata on nested words. The idea is to remove all rules and states from an automaton that are not used to recognize any word satisfying the schema. Schema-based cleaning can be based on the accessible states of the product of the automaton with the schema. While this product may be larger than the automaton, the schema-based cleaning will always be smaller.

$$\frac{q \in I^A \quad s \in I^S}{(q,s) \in I^{A\times S} \qquad (q,s) \in \mathcal{Q}^{A\times S}} \qquad \frac{q \in F^A \quad s \in F^S \quad (q,s) \in \mathcal{Q}^{A\times S}}{(q,s) \in F^{A\times S}}$$

$$\frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (q_1,s_1) \in \mathcal{Q}^{A\times S}}{(q_1,s_1) \xrightarrow{a} (q_2,s_2) \in \Delta^{A\times S} \quad (q_2,s_2) \in \mathcal{Q}^{A\times S}}$$

Figure 7: Accessible product $A \times S = (\Sigma, \mathcal{Q}^{A\times S}, I^{A\times S}, F^{A\times S}, \Delta^{A\times S})$.

For illustration, the schema-based cleaning of NFA $\det(A_0)$ in Figure 4 with respect to schema *words-one*$^x_\Sigma$ is given in Figure 5. The only words recognized by both $\det(A_0)$ and *words-one*$^x_\Sigma$ are x and xa. For recognizing these two words, the automaton $\det(A_0)$ does not need states 2 and 3, so they can be removed with all their transitions rules. Thereby, the word xxa violating the schema is no more recognized after schema-based cleaning, while it was recognized by $\det(A_0)$. Furthermore, note that the state 0 needs no more to be final after schema-based cleaning. Therefore the word ε, which is recognized by the automaton but not by the schema, is no more recognized after schema-based cleaning. So schema-based cleaning may change the language of the automaton but only outside of the schema.

Interestingly, the NFA A_0 in Figure 3 is schema-clean for schema *words-one*$^x_\Sigma$ too, even though it is not perfect, in that it recognizes the words ε and xxa which are rejected by the schema. The reason is that for recognizing the words x and xa, which both satisfy the schema, all 3 states and all 4 transition rules of A_0 are needed. In contrast, we already noticed that the accessible determinization $\det(A_0)$ in Figure 4 is not schema-clean for schema *words-one*$^x_\Sigma$. This illustrates that accessible determinization does not always preserve schema-cleanliness. In other words, schema-based cleaning may have a stronger cleaning effect after determinization than before.

The schema-based cleaning of an automaton can be defined based on the accessible product of the automaton with the schema. The accessible product $A \times S$ of two NFAs A and S with alphabet Σ is defined in Figure 7. This is the usual product, except that only accessible states are admitted. Clearly, $\mathcal{L}(A \times S) = \mathcal{L}(A) \cap \mathcal{L}(S)$. Let $\Pi_A(A \times S)$ be obtained from the accessible product by projecting away the second component. The schema-based cleaning of A with respect to schema S is this projection.

Definition 3. $scl_S(A) = \Pi_A(A \times S)$.

The fact that $A \times S$ is restricted to accessible states matches our intuition that all states of $scl_S(A)$ can be used to read some word in $\mathcal{L}(A)$ that satisfies schema S. This can be proven formally under the condition that all states of $A \times S$ are also co-accessible. Clearly, $scl_S(A)$ is obtained from A by removing states, initial states, final states, and transitions rules. So it is smaller or equal in size $|scl_S(A)| \leq |A|$ and language $\mathcal{L}(scl_S(A)) \subseteq \mathcal{L}(A)$. Still, schema-based cleaning preserves the language within the schema.

Proposition 4 ([19]). $\mathcal{L}(A) \cap \mathcal{L}(S) = \mathcal{L}(scl_S(A)) \cap \mathcal{L}(S)$.

Schema-clean deterministic automata may still not be perfect, in that they may recognize some words outside the schema. This happens for DFAs if some state of is reached, both, by a word satisfying the schema and another word that does not satisfy the schema. An example for a DFA that is schema-clean but not perfect for *words-one*$^x_\Sigma$ is given in Figure 8. It is not perfect since it accepts the non V-structure $xaxa$. The problem is that state 1 can be reached by the words a and xa, so one cannot infer from being in state 1 whether some x was read or not. If one wants to avoid this, one can use the accessible product of the DFA with the schema instead. In the example, this yields the DFA in Figure 9 that is schema-clean and perfect for *words-one*$^x_\Sigma$.

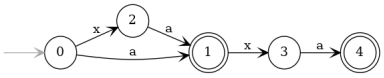


Figure 8: A DFA that is schema-clean but not perfect for $words\text{-}one^{x}_{\Sigma}$.

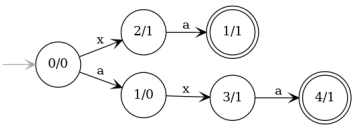


Figure 9: The accessible product with $words\text{-}one^{x}_{\Sigma}$ is schema-clean and perfect for $words\text{-}one^{x}_{\Sigma}$.

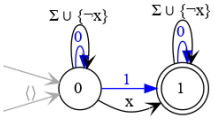


Figure 10: The dSHA one^{x}_{Σ} with alphabet $\Sigma \uplus \{x, \neg x\}$.

Proposition 5 (Folklore)**.** *For any two* DFA*s A and S with alphabet Σ the accessible product $A \times S$ can be computed in expected amortized time $O(|2^{A\times S}||\Sigma| + |A| + |S|)$.*

Proof. An algorithm to compute the fixed points of the inference rules for the accessible product $A \times S$ in Figure 7 can be organized such that only accessible states are considered (similarly to semi-naive datalog evaluation). This algorithm is presented in Figure 11. It dynamically generates the set of rules *Rules* by using perfect dynamic hashing [11]. Testing set membership is in time $O(1)$ and the addition of elements to the set is in expected amortized time $O(1)$. The algorithm uses a stack, *Agenda*, to memoize all new pairs $(q_1, s_1) \in 2^{A\times S}$ that need to be processed, and a hash set *Store* that saves all processed states $2^{A\times S}$. We aim not to push the same pair more than once in the *Agenda*. For this, membership to the *Store* is checked before an element is pushed to the *Agenda*. For each pair popped from the stack *Agenda*, the algorithm does the following: for each letter $a \in \Sigma$ it computes the sets $Q = \{q_2 \mid q_1 \xrightarrow{a} q_2 \in \Delta^A\}$ and $R = \{s_2 \mid s_1 \xrightarrow{a} s_2 \in \Delta^S\}$ and then adds the subset of states of $Q \times R$ that were not stored in the hash set *Store* to the agenda. Since A and S are deterministic, there is at most one such pair, so the time for treating one pair on the agenda is in expected amortized time $O(|\Sigma|)$. The overall number of elements in the agenda will be $|2^{A\times S}|$. Note that Q and R can be computed in $O(1)$ after preprocessing A and S in time $O(|A| + |S|)$. Therefore, we will have a total time of the algorithm in $O(|2^{A\times S}||\Sigma| + |A| + |S|)$. □

Corollary 6. *For any two* DFA*s A and S with alphabet Σ schema-based cleaning $scl_S(A)$ can be computed in expected amortized time $O(|2^{A\times S}||\Sigma| + |A| + |S|)$.*

Proof. By Definition 3 it is sufficient to compute the projection of the accessible product $A \times S$. By Proposition 5 the product can be computed in time $O(|2^{A\times S}||\Sigma| + |A| + |S|)$. Its size cannot be larger than its computation time. The projection can be computed in linear time in the size of $A \times S$, so the overall time is in $O(|2^{A\times S}||\Sigma| + |A| + |S|)$ too. □

5 Schema-Based Determinization

Schema-based cleaning after determinization becomes impossible in practice if the automaton obtained by determinization is too big. We therefore show next how to integrate schema-based cleaning into automata determinization directly.

The schema-based determinization of A with respect to schema S extends on accessible determinization $\det(A)$. The idea is to run the schema S in parallel with $\det(A)$, in order to keep only those state $Q \in 2^{\det(A)}$ that can be aligned to some state $s \in 2^S$. In this case we write $Q \sim s$.

```
1  fun A×S =
2    let Store = hashset.new(∅) and Agenda = list.new() and Rules = hashset.new(∅)
3    if init^A = {q0} and init^S = {s0} then Agenda.add((q0,s0))
4    while Agenda.notEmpty() do
5      let (q1,s1) = Agenda.pop()
6      for a ∈ Σ do
7        let Q = {q2 | q1 -a-> q2 ∈ Δ^A}  R = {s2 | s1 -a-> s2 ∈ Δ^S}
8        for q2 ∈ Q and s2 ∈ R do
9          Rules.add((q1,s1) -a-> (q2,s2))
10         if not Store.member((q2,s2))
11           then Store.add((q2,s2))  Agenda.push((q2,s2))
12   let init^{A×S} = {(q0,s0) | (q0,s0) ∈ Store} and F^{A×S} = {(q,s) | (q,s) ∈ Store, q ∈ F^A, s ∈ F^S}
13   return (Σ, Store.toSet(), Rules.toSet(), init^{A×S}, F^{A×S})
```

Figure 11: An algorithm computing the accessible product of DFAs A and S.

$$\frac{Q \in I^{det(A)} \quad I^S = \{s\}}{Q \in I^{det_S(A)} \quad Q \sim s} \qquad \frac{Q \sim s}{Q \in \mathcal{Q}^{det_S(A)}} \qquad \frac{Q \in F^{\det(A)} \qquad s \in F^S \qquad Q \sim s}{Q \in F^{det_S(A)}}$$

$$\frac{Q \xrightarrow{a} Q' \in \Delta^{det(A)} \qquad Q \sim s \qquad s \xrightarrow{a} s' \in \Delta^S}{Q \xrightarrow{a} Q' \in \Delta^{det_S(A)} \qquad Q' \sim s'}$$

Figure 12: Schema-based determ. $det_S(A) = (\Sigma, \mathcal{Q}^{det_S(A)}, \Delta^{det_S(A)}, I^{det_S(A)}, F^{det_S(A)})$.

The schema-determinization $det_S(A)$ is defined in Figure 12. The automaton $det_S(A)$ permits to go from any subset $Q \in \mathcal{Q}^{\det(A)}$ and letter $a \in \Sigma$ to the set of states $Q' = a^{\Delta^{\det(A)}}(Q)$, under the condition that there exists schema states $s, s' \in \mathcal{Q}^S$ such that $Q \sim s$ and $s \xrightarrow{a} s'$. In this case $Q' \sim s'$ is inferred.

Theorem 1 (Correctness)**.** *$det_S(A) = scl_S(\det(A))$ for any* NFA *A and* DFA *S with the same alphabet.*

The theorem states that schema-based determinization yields the same result as accessible determinization followed by schema-based cleaning.

For the correctness proof we collapse the two systems of inference rules for accessible products and projection into a single rule system. This yields the rule systems for schema-based cleaning in Figure 13. The rules there define the automaton $\widehat{scl}_S(A)$, that we annotate with a hat, in order to distinguish it from the previous automaton $scl_S(A)$. The rules also infer judgements $(q,s) \in \mathcal{Q}^{A\widehat{\times}S}$ that we distinguish by a hat from the previous judgments $(q,s) \in \mathcal{Q}^{A\times S}$ of the accessible product. The next proposition shows that the system of collapsed inference rules indeed redefines the schema-based cleaning.

Proposition 7. *For any two* NFA*s A and S with the same alphabet:*

$$scl_S(A) = \widehat{scl}_S(A) \quad \text{and} \quad \mathcal{Q}^{A\times S} = \mathcal{Q}^{A\widehat{\times}S}$$

Proof of Correctness Theorem 1. Instantiating the system of collapsed rules for schema-based cleaning from Figure 13 with $\det(A)$ for A yields the rule system in Figure 15. We can identify the instantiated collapsed system for $\widehat{scl}_S(det(A))$ with that for $det_S(A)$ in Figure 12, by identifying the judgements $(Q,s) \in \mathcal{Q}^{det(A)\widehat{\times}S}$ with judgments $Q \sim s$. After renaming the predicates, the inference rules for the corresponding judgments are the same. Hence $\widehat{scl}_S(det(A)) = det_S(A)$, so that Proposition 7 implies $scl_S(det(A)) = det_S(A)$. □

$$\frac{q \in I^A \quad s \in I^S}{q \in I^{\widehat{scl}_S(A)} \quad (q,s) \in \mathcal{Q}^{A \widehat{\times} S}} \qquad \frac{q \in F^A \quad s \in F^S \quad (q,s) \in \mathcal{Q}^{A \widehat{\times} S}}{q \in F^{\widehat{scl}_S(A)}}$$

$$\frac{(q,s) \in \mathcal{Q}^{A \widehat{\times} S}}{q \in \mathcal{Q}^{\widehat{scl}_S(A)}} \qquad \frac{q_1 \xrightarrow{a} q_2 \in \Delta^A \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (q_1,s_1) \in \mathcal{Q}^{A \widehat{\times} S}}{q_1 \xrightarrow{a} q_2 \in \Delta^{\widehat{scl}_S(A)} \quad (q_2,s_2) \in \mathcal{Q}^{A \widehat{\times} S}}$$

$$\widehat{scl}_S(A) = (\Sigma, \mathcal{Q}^{\widehat{scl}_S(A)}, \Delta^{\widehat{scl}_S(A)}, I^{\widehat{scl}_S(A)}, F^{\widehat{scl}_S(A)})$$

Figure 13: A collapsed rule systems for schema-based cleaning $\widehat{scl}_S(A)$.

```
1  fun detS(A,S) =
2    let Store = hashset.new(∅) and Agenda = list.new() and Rules = hashset.new(∅)
3    if init^A ≠ ∅ and init^S = {s_0} then Agenda.add(init^A ∼ s_0)
4    while Agenda.notEmpty() do
5      let (Q_1 ∼ s_1) = Agenda.pop()
6      for a ∈ Σ do
7        let P = {Q_2 | Q_1 -a-> Q_2 ∈ Δ^det(A)} and R = {s_2 | s_1 -a-> s_2 ∈ Δ^S}
8        for Q_2 ∈ P and s_2 ∈ R do Rules.add(Q_1 -a-> Q_2)
9          if not Store.member(Q_2 ∼ s_2)
10           then Store.add(Q_2 ∼ s_2) Agenda.push(Q_2 ∼ s_2)
11   let init^{det_S(A)} = {Q | Q ∼ s ∈ Store, Q ∩ init^A ≠ ∅} and F^{det_S(A)} = {Q | Q ∼ s ∈ Store, Q ∩ F^A ≠ ∅}
12   return (Σ, Store.toSet(), Rules.toSet(), init^{det_S(A)}, F^{det_S(A)})
```

Figure 14: An algorithm for schema-based determinization $det_S(A)$ of an NFA A and a DFA schema S.

Proposition 8. *The schema-based determinization $det_S(A)$ for a* NFA *A and a* DFA *S over Σ can be computed in expected amortized time $O(|\mathcal{Q}^{\det(A)\times S}||\Sigma| + |\mathcal{Q}^{det_S(A)}||\Delta^A| + |A| + |S|)$.*

Proof. An algorithm computing the fixed points of the inference rules of schema-based determinization from Figure 12 is given in Figure 14. It refines the algorithm computing the accessible product with on-the-fly determinization and projection.

On the stack *Agenda*, the algorithm stores alignments $Q \sim s$ such that $(Q,s) \in \mathcal{Q}^{\det(A)\times S}$ that were not considered before. Transition rules of $det_S(A)$ are collected in hash set *Rules*, using the dynamic perfect hashing aforementioned. The alignments $Q_1 \sim s_1$ popped from the agenda are processed as follows: For any letter $a \in \Sigma$, the sets $R = \{Q_2 \mid Q_1 \xrightarrow{a} Q_2 \in \Delta^{\det(A)}\}$ and $P = \{s_2 \mid s_1 \xrightarrow{a} s_2 \in \Delta^S\}$ are computed. One then pushes all new pairs $Q_2 \sim s_2$ with $Q_2 \in P$ and $s_2 \in R$ into the agenda, and adds $Q_1 \xrightarrow{a} Q_2$ to the set *Rules*. Since S and $\det(A)$ are deterministic there is at most one pair $(Q,s) \in P \times R$ for Q_1 and s_1. So the time for treating one pair on the agenda is in $O(|\Sigma|)$ plus the time for building the needed transition rules of $\det(A)$ from Δ^A on the fly. The time for the on the fly computation of transition rules of $\det(A)$ is in time $O(|\mathcal{Q}^{det_S(A)}||\Delta^A|)$. The overall number of pairs on the agenda is at most $|\mathcal{Q}^{\det(A)\times S}|$ so the main while loop of the algorithm requires time in $O(|\mathcal{Q}^{\det(A)\times S}||\Sigma|)$ apart from on the fly determinization. □

By Proposition 2, computing $\det(A)$ requires time $O(|\mathcal{Q}^{det(A)}|\ |\Delta^A| + |A|)$. Therefore, with Proposition 5, the accessible product $\det(A) \times S$ can be computed from A and S in time $O(|\mathcal{Q}^{\det(A)\times S}||\Sigma| + |\mathcal{Q}^{det(A)}|\ |\Delta^A| + |A| + |S|)$. Since $\mathcal{Q}^{det_S(A)} \subseteq \mathcal{Q}^{\det(A)}$ the proposition shows that schema-based determinization is at most as efficient in the worst case as accessible determinization followed by schema-based cleaning. If $|\mathcal{Q}^{\det(A)\times S}||\Sigma| < |\mathcal{Q}^{\det(A)}||\Delta^A|$ then it is more efficient, since schema-based determinization

$$\frac{Q \in I^{det(A)} \quad s \in I^S}{Q \in I^{\widehat{scl_S}(det(A))} \quad (Q,s) \in \mathscr{Q}^{det(A)\widehat{\times}S}} \qquad \frac{Q \in F^{det(A)} \quad s \in F^S \quad (Q,s) \in \mathscr{Q}^{det(A)\times S}}{Q \in F^{\widehat{scl_S}(det(A))}}$$

$$\frac{(Q,s) \in \mathscr{Q}^{det(A)\widehat{\times}S}}{Q \in \mathscr{Q}^{\widehat{scl_S}(det(A))}} \qquad \frac{Q_1 \xrightarrow{a} Q_2 \in \Delta^{det(A)} \quad s_1 \xrightarrow{a} s_2 \in \Delta^S \quad (Q_1,s_1) \in \mathscr{Q}^{det(A)\times S}}{Q_1 \xrightarrow{a} Q_2 \in \Delta^{\widehat{scl_S}(det(A))} \quad (Q_2,s_2) \in \mathscr{Q}^{det(A)\widehat{\times}S}}$$

$$\widehat{scl_S}(det(A)) = (\Sigma, \mathscr{Q}^{\widehat{scl_S}(det(A))}, \Delta^{\widehat{scl_S}(det(A))}, I^{\widehat{scl_S}(det(A))}, F^{\widehat{scl_S}(det(A))})$$

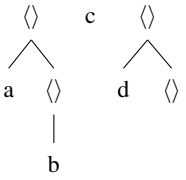
Figure 15: Instantiation of the collapsed rules for schema-based cleaning from Figure 13 with $\det(A)$.

avoids the computation of $\det(A)$ all over. Instead, it only computes the accessible product $\det(A) \times S$, which may be way smaller, since exponentially many states of $\det(A)$ may not be aligned to any state of S. Sometimes, however, the accessible product may be bigger. In this case, schema-based determinization may be more costly than pure accessible determinization, not followed by schema-based cleaning.

6 Stepwise Hedge Automata for Nested Words

We next recall SHAs [19] for defining languages of nested words, regular schemas and queries. Nested words generalize on words by adding parenthesis that must be well-nested. While containing words natively, they also generalize on unranked trees, and hedges. We restrict ourselves to nested words with a single pair of opening and closing parenthesis $\langle$ and $\rangle$. Nested words over a finite alphabet Σ of internal letters have the following abstract syntax.

$$w, w' \in \mathscr{N}_\Sigma \quad ::= \quad \varepsilon \mid a \mid \langle w \rangle \mid w \cdot w' \qquad \text{where } a \in \Sigma$$



We assume that concatenation $\cdot$ is associative and that the empty word ε is a neutral element, that is $w \cdot (w' \cdot w'') = (w \cdot w') \cdot w''$ and $\varepsilon \cdot w = w = w \cdot \varepsilon$. Nested words can be identified with hedges, i.e., words of unranked trees and letters from Σ. Seen as a graph, the inner nodes are labeled by the tree constructor $\langle\rangle$ and the leafs by symbols in Σ or the tree constructor. For instance $\langle a \cdot \langle b \rangle \cdot \varepsilon \rangle \cdot c \cdot \langle d \cdot \langle \varepsilon \rangle \rangle$ corresponds to the hedge on the right. A nested word of type *tree* has the form $\langle h \rangle$. Note that dangling parentheses are ruled out and that labeled parentheses can be simulated by using internal letters. *XML* documents are labeled unranked trees, for instance: $\langle a\ name = \text{“}uff\text{”} \rangle \langle b \rangle isgaga \langle d/\rangle \langle /b \rangle \langle c/\rangle \langle /a \rangle$. Labeled unranked trees satisfying the *XML* data model can be represented as nested words over an alphabet that contains the *XML* node-types $(elem, attr, text, \ldots)$, the *XML* names of the document $(a, \ldots, d, name)$, and the characters of the data values, say UTF8. For the above example, we get the nested word $\langle elem \cdot a \cdot \langle attr \cdot name \cdot u \cdot f \cdot f \rangle \langle elem \cdot b \cdot \langle text \cdot i \cdot s \cdot g \cdot a \cdot g \cdot a \rangle \langle elem \cdot d \rangle \rangle \langle elem \cdot c \rangle \rangle$

Definition 9. *A* SHA *is a tuple* $A = (\Sigma, \mathscr{Q}, \Delta, I, F)$ *where* $\Delta = (\Delta', @^\Delta, \langle\rangle^\Delta)$ *such that* $(\Sigma, \mathscr{Q}, \Delta', I, F)$ *is a* NFA, $\langle\rangle^\Delta \subseteq \mathscr{Q}$ *is a set of tree initial states and* $@^\Delta \subseteq \mathscr{Q}^3$ *a set of apply rules.*

SHAs can be drawn as graphs while extending on the graphs of NFAs. A tree initial state $q \in \langle\rangle^\Delta$ is drawn as a node $\xrightarrow{\langle\rangle}(q)$ with an incoming tree arrow. An applyrule $(q_1, q, q_2) \in @^\Delta$ is drawn as a blue edge $(q_1) \xrightarrow{q} (q_2)$ that is labeled by a state $q \in \mathscr{Q}$ rather than a letter $a \in \Sigma$. It states that a nested word in state q_1 can be extended by a tree in state q and become a nested word in state q_2.

For instance, the SHA one^x_Σ is drawn graphically in Figure 10. It accepts all nested words over $\Sigma \uplus \{x, \neg x\}$ that contain exactly one occurrence of letter x. Compared to the NFA $words\text{-}one^x_{\Sigma \uplus \{\neg x\}}$ from

$$\frac{\langle\!\rangle^{\Delta^A} \neq \emptyset}{\langle\!\rangle^{\Delta^A} \in \mathscr{Q}^{\det(A)}} \qquad \frac{Q_1 \in \mathscr{Q}^{\det(A)} \qquad Q_2 \in \mathscr{Q}^{\det(A)} \qquad Q' = \{q' \in \mathscr{Q}^A \mid q_1 @ q_2 \to q' \in \Delta^A,\ q_1 \in Q_1, q_2 \in Q_2\} \neq \emptyset}{Q_1 @ Q_2 \to Q' \in \Delta^{det(A)} \qquad Q' \in \mathscr{Q}^{\det(A)}}$$

Figure 16: Accessible determinization $\det(A)$ lifted from NFAs to SHAs.

Figure 6, the SHA one^x_Σ contains three additional apply rules $(0,0,0)$, $(0,1,1)$, $(1,0,1) \in @^{\Delta^{one^x_\Sigma}}$ for reading the states assigned to subtrees. The state 0 is chosen as the single tree initial state.

Transitions for NFAs on words can be lifted to transitions for SHAs of the form $q \xrightarrow{w} q'$ wrt Δ where $w \in \mathscr{N}_\Sigma$ and $q, q' \in \mathscr{Q}$. For this, we add the following inference rule to the previous rules for NFAs:

$$\frac{q' \in \langle\!\rangle^\Delta \qquad q' \xrightarrow{w} q \text{ wrt } \Delta \qquad (q_1, q, q_2) \in @^\Delta}{q_1 \xrightarrow{\langle w \rangle} q_2 \text{ wrt } \Delta}$$

The rule says that a tree $\langle w \rangle$ can transit from a state q_1 to a state q_2 if there is an apply rule $(q_1, q, q_2) \in @^\Delta$ so that w can transit from some tree initial state $q' \in \langle\!\rangle^\Delta$ to q. Otherwise, the language $\mathscr{L}(A)$ of nested words accepted by a SHA A is defined as in the case of NFAs.

Definition 10. *A* SHA $(\Sigma, \mathscr{Q}, \Delta, I, F)$ *is* deterministic *or equivalently a d*SHA *if it satisfies:*

- *I and $\langle\!\rangle^\Delta$ both contain at most one element,*
- *a^Δ is a partial function from $\mathscr{Q}$ to $\mathscr{Q}$ for all $a \in \Sigma$, and*
- *$@^\Delta$ is a partial function from $\mathscr{Q} \times \mathscr{Q}$ to $\mathscr{Q}$.*

Note that if A is a dSHA and $\Delta = (\Delta', @^\Delta, \langle\!\rangle^\Delta)$ then $A' = (\Sigma, \mathscr{Q}, \Delta', I, F)$ is a DFA. Conversely any DFA A' defines a dSHA with $@^\Delta = \emptyset$ and $I = \emptyset$. For instance, the SHA one^x_Σ in Figure 10 contains the DFA $words\text{-}one^x_{\Sigma \uplus \{\neg x\}}$ from Figure 6 with Σ instantiated by $\Sigma \uplus \{x\}$.

A schema for nested words over Σ is a dSHA over Σ. Note that schemas for nested words generalize over schemas of words, since dSHAs generalize on DFAs. The rules for the accessible determinization $\det(A)$ of a SHA A in Figure 16 extend on those for NFAs in Figure 1. As for words, $\det(A)$ is always determinstic, recognizes the same language as A, and contains only accessible states. The complexity of accessible determinization in case of SHA go similarly to DFA, however, the apply rules will introduce quadratic factor in the number of states.

Proposition 11. *The accessible determinization of a* SHA *can be computed in expected amortized time* $O(|\mathscr{Q}^{\det(A)}|^2\ |\Delta^A| + |A|)$.

The notions of monadic query functions $\mathbf{Q}$ can be lifted from words to nested words, so that it selects nodes of the graph of a nested word. For this, we have to fix one of manner possible manners to define identifiers for these nodes. The set of nodes of a nested word w is denoted by $nod(w) \subseteq \mathbb{N}$.

For indicating the selection of node $\pi \in nod(w)$, we insert the variable x into the sequence of letters following the opening parenthesis of π. If we don't want to select π, we insert the letter $\neg x$ instead. For any nested word w with alphabet Σ, the nested word $w[\pi/x]$ obtained by insertion of x or $\neg x$ at a node $\pi \in nod(w)$ has alphabet $\Sigma \uplus \{x, \neg x\}$. As before, we define $L_{\mathbf{Q}} = \{w * [\pi/x] \mid w \in \mathscr{N}_\Sigma, \pi \in \mathbf{Q}(w)\}$.

The notion of a query automata can now be lifted from words to nested words straightforwardly: a query automaton for nested words over Σ is a SHA A with alphabet $\Sigma \cup \{x, \neg x\}$. It defines the unique total query $\mathbf{Q}$ such that $L_{\mathbf{Q}} = \mathscr{L}(A) \cap \mathscr{L}(one^x_\Sigma)$.

$$\frac{q \in \langle\rangle^{\Delta^A} \quad s \in \langle\rangle^{\Delta^S}}{(q,s) \in \langle\rangle^{\Delta^{A\times S}} \quad (q,s) \in \mathcal{Q}^{A\times S}} \qquad \frac{\begin{array}{cc}(q_1,s_1) \in \mathcal{Q}^{A\times S} & q_1@q \to q_2 \in \Delta^A \\ (q,s) \in \mathcal{Q}^{A\times S} & s_1@s \to s_2 \in \Delta^S\end{array}}{(q_1,s_1)@(q,s) \to (q_2,s_2) \in \Delta^{A\times S} \quad (q_2,s_2) \in \mathcal{Q}^{A\times S}}$$

Figure 17: Lifting accessible products to SHAs.

$$\frac{\langle\rangle^{\Delta^S} = \{s\}}{\langle\rangle^{\Delta^A} \in \langle\rangle^{\Delta^{det_S(A)}} \quad \langle\rangle^{\Delta^A} \sim s} \qquad \frac{\begin{array}{ccc}s_1@s_2 \to s' \in \Delta^S & Q_1 \sim s_1 & Q_2 \sim s_2 \\ Q_1@Q_2 \to Q' \in \Delta^{det(A)} & & \end{array}}{Q_1@Q_2 \to Q' \in \Delta^{det_S(A)} \qquad Q' \sim s'}$$

Figure 18: Extension of schema-based determinization to SHAs.

7 Schema-Based Determinization for SHAs

We can lift all previous algorithms from NFAs to SHAs while extending the system of inference rules. The additional rules concern tree initial states, that work in analogy to initial states, and also apply rules that works similarly as internal rules. The new inference rules for accessible products $A \times S$ are given in Figure 17 . As before we define $scl_S(A) = \Pi_A(A \times S)$. The rules for schema-based determinization $det_S(A)$ are extended in Figure 18. The complexity upper bound, however, now becomes quadratic even with fixed alphabet:

Proposition 12. *If A and S are d*SHA*s then the accessible product $A \times S$ and the schema-based cleaning $scl_S(A)$ can be computed in expected amortized time $O(|\mathcal{Q}^{A\times S}|^2 + |\mathcal{Q}^{A\times S}||\Sigma| + |A| + |S|)$.*

Theorem 2 (Correctness)**.** *$det_S(A) = scl_S(\det(A))$ for any* SHA *A and dSHA S with the same alphabet.*

Proposition 13. *The schema-based determinization $det_S(A)$ of a* SHA *A with respect to a d*SHA *S can be computed in expected amortized time $O(|\mathcal{Q}^{\det(A)\times S}|^2 + |\mathcal{Q}^{\det(A)\times S}|\ |\Sigma| + |\mathcal{Q}^{det_S(A)}|^2\ |\Delta^A| + |A| + |S|)$.*

The proof of Theorem 2 extends on that for NFAs (Theorem 1) in a direct manner. Proposition 13 follows the result in Proposition 8 with an additional quadratic factor in the size of states of the product $det(A) \times S$ and the states of the schema-based determinized automaton. This is always due to the apply rules of type $\mathcal{Q}^3$. By Propositions 11 and 12, computing $scl_S(det(A))$ by schema-based cleaning after accessible determinization needs time in $O(|\mathcal{Q}^{\det(A)\times S}|^2 + |\mathcal{Q}^{\det(A)\times S}|\ |\Sigma| + |\mathcal{Q}^{\det(A)}|^2\ |\Delta^A| + |A| + |S|)$. This complexity bound is similar to that of schema-based determinization from Proposition 13. Since $\mathcal{Q}^{det_S(A)} \subseteq \mathcal{Q}^{\det(A)}$, Proposition 13 shows that the worst case time complexity of schema-based determinization is never worse than for schema-based cleaning after determinization.

8 Experiments

In this section, we present an experimental evaluation of the sizes of the automata produced by the different determinization methods. For this, we consider a scalable family of SHAs that is compiled from the following scalable family of XPATH queries where n and m are natural numbers.

```
(Qn.m)    //*[self::a0 or ... or self::an]
          [descendant::*[self::b0 or ... or self::bm]]
```

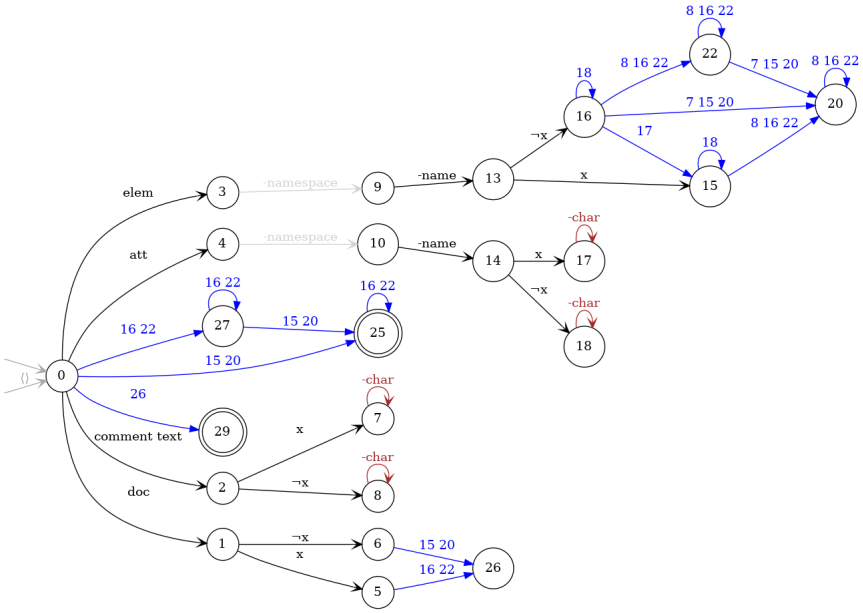


Figure 19: A schema for the intersection of *XML* data model with one^x.

Query `Qn.m` selects all elements of an *XML* document, that are named by either of `a0`, ..., `an` and have some descendant element named by either of `b1`, ..., `bm`. We compile those XPATH queries to SHAs based on the compiler from [19]. As schema S, we chose the product of the dSHA one^x with a dSHA for the XML data model given in Figure 19. Beside the concepts presented above, this SHA also has typed else rules. Actually, we use a richer class of SHAs in the experiments, which is converted back into the class of the paper when showing the results (except for else rules and typed else rules).

The results of our experiments are summarized in Table 20. For each automaton we present two numbers, size(#states), its size and the number of its states. Unless specified otherwise, we use a timeout of 1000 seconds whenever calling some determinization algorithm. Fields of the table are left blank if an exception was raised. This happens when the determinization algorithm reached the timeout, the memory was filled, or the stack overflowed. We conducted all the experiments on a Dell laptop with the following specs: Intel® Core™ i7-10875H CPU @ 2.30 GHz,16 cores, and 32 GB of RAM.

The first column A of Table 20 reports on the SHAs obtained from the queries Qn.m, by the compiler from [19] that is written in XSLT. The second column $det(A)$ is obtained from SHA A by accessible determinization. The blank cell in column $det(A)$ for query Q4.4 was raised by a timeout of the determinization algorithm. As one can see, this happens for all larger pairs (n,m). Furthermore, it appears that the sizes of the automata $det(A)$ grow exponentially with $n+m$.

In the third column $det(A \times S)$, the determinization of the product is presented. It yields much smaller automata than with $det(A)$. For Q4.3 for instance, $det(A)$ has size 53550 (2161) while $det(A \times S)$ has size 5412 (438). The computation continues successfully until Q6.4. For the larger queries Q6.5 and Q6.6, our determinizer runs out of memory. The fourth column $det_S(A)$ reports on schema-based determinization. For Q4.3 for instance we obtain 3534 (329). Here and in all given examples, both measures are always smaller for $det_S(A)$ than for $det(A \times S)$. While this may not always be the case, but both approaches yield decent results generally. The numbers for the $det_S(A)$ for Q6.6 are marked in gray, since its computation took around one hour, so we obtain it only when ignoring the timeout. In contrast to $det(A \times S)$, however, the computation of $det_S(A)$ did not run out of memory though. The fifth

	A	$det(A)$	$det(A \times S)$	$det_S(A)$	$scl_S($ $det(A))$	$mini(det$ $(A \times S))$	$mini($ $det_S(A))$
Q2.1	166 (67)	1380 (101)	540 (92)	284 (53)	284 (53)	160 (43)	73 (20)
Q2.2	199 (79)	3635 (214)	1488 (167)	830 (106)		162 (43)	75 (20)
Q2.3	232 (91)	9574 (471)	4174 (334)	2424 (227)		164 (43)	77 (20)
Q2.4	265 (103)	24813 (1052)	11502 (713)	6826 (504)		166 (43)	79 (20)
Q4.1	240 (95)	8020 (435)	710 (116)	418 (75)		164 (43)	77 (20)
Q4.2	287 (111)	20945 (968)	1944 (215)	1220 (152)		166 (43)	79 (20)
Q4.3	334 (127)	53550 (2161)	5412 (438)	3534 (329)		168 (43)	81 (20)
Q4.4	381 (143)		14794 (945)	9856 (734)		170 (43)	83 (20)
Q6.1	314 (123)	48212 (2113)	880 (140)	552 (97)		168 (43)	81 (20)
Q6.2	375 (143)		2400 (263)	1610 (198)		170 (43)	83 (20)
Q6.3	436 (163)		6650 (542)	4644 (431)		172 (43)	85 (20)
Q6.4	497 (183)		18086 (1177)	12886 (964)			87 (20)
Q6.5	558 (203)			34376 (2169)			
Q6.6	619 (223)			88666 (4862)			

Figure 20: Statistics of automata for XPATH queries: size(#states)

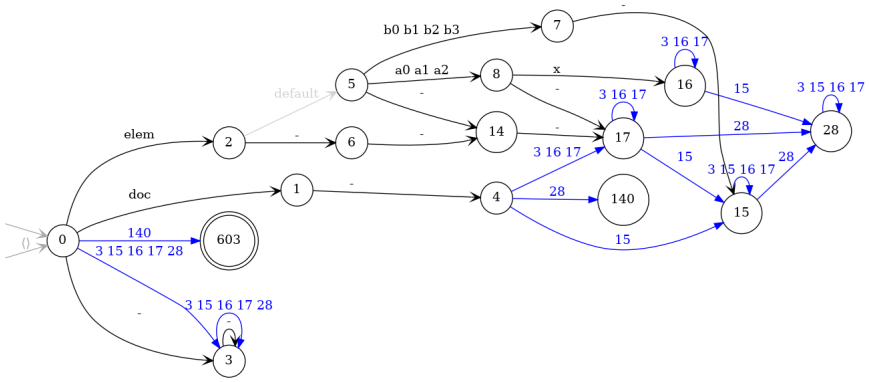


Figure 21: The automaton $mini(det_S(A))$ of the query Q3.4.

column $scl_S(\text{det}(A))$ contains the schema-based cleaning of $\text{det}(A)$. This automaton is equal to $det_S(A)$ by Correctness Theorem 2. Nevertheless, this cell is left blank in all but the smallest case $Q2.1$, since our datalog implementation of schema-based cleaning quickly runs out of memory for automata with many states. The time in seconds that for determinization in $det(A \times S)$ and $det_S(A)$ grows in dependence of the size of the output from 0.9 seconds until passing over the timeout.

In the last two columns for $mini(det(A \times S))$ and $mini(detS(A))$ we report the sizes of the minimization of $det(A \times S))$ and $det_S(A)$. It turns out that $mini(det_S(A))$ is always smaller than $mini(det(A \times S))$, if both can be computed successfully. An example of $mini(det_S(Q3.4))$ is shown in Figure 21.

Conclusion and Future Work

We presented an algorithm for schema-based determinization for SHAs and proved that it always produces the same results as determinization followed by schema-based cleaning. We argued why schema-based determinization is often way more efficient than standard determinization, and why it is close in efficiency to the determinization of the schema-product. The statements are supported by upper complexity bounds and experimental evidence. The experimental results of the present paper are enhanced by follow up work [1]. They show that one can indeed obtain small deterministic automata based on schema-based determinization of stepwise hedge automata for all regular XPATH queries in practice. We hope that these automata are useful in the future for experiments with query answering.

References

[1] Antonio Al Serhali & Joachim Niehren (2022): *A Benchmark Collection of Deterministic Automata for XPath Queries*. In: *XML Prague 2022*, Prague, Czech Republic. Available at `https://hal.inria.fr/hal-03527888`.

[2] Rajeev Alur (2007): *Marrying Words and Trees*. In: *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ACM-Press, pp. 233–242. Available at `http://dx.doi.org/10.1145/1265530.1265564`.

[3] Rajeev Alur & P. Madhusudan (2004): *Visibly pushdown languages*. In László Babai, editor: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, ACM, pp. 202–211, doi:10.1145/1007352.1007390.

[4] Rajeev Alur & P. Madhusudan (2009): *Adding nesting structure to words*. *Journal of the ACM* 56(3), pp. 1–43. Available at `http://doi.acm.org/10.1145/1516512.1516518`.

[5] Mikolaj Bojanczyk & Igor Walukiewicz (2008): *Forest algebras*. In Jörg Flum, Erich Grädel & Thomas Wilke, editors: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*, *Texts in Logic and Games* 2, Amsterdam University Press, pp. 107–132.

[6] Burchard von Braunmühl & Rutger Verbeek (1985): *Input Driven Languages are Recognized in log n Space*. In Marek Karplnski & Jan van Leeuwen, editors: *Topics in the Theory of Computation*, *North-Holland Mathematics Studies* 102, North-Holland, pp. 1 – 19, doi:10.1016/S0304-0208(08)73072-X.

[7] J. Richard Büchi (1960): *Weak Second-Order Arithmetic and Finite Automata*. *Mathematical Logic Quarterly* 6(1-6), pp. 66–92, doi:10.1002/malq.19600060105. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105.

[8] Julien Carme, Joachim Niehren & Marc Tommasi (2004): *Querying Unranked Trees with Stepwise Tree Automata*. In Vincent van Oostrom, editor: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, *Lecture Notes in Computer Science* 3091, Springer, pp. 105–118, doi:10.1007/978-3-540-25979-4_8.

[9] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison & Marc Tommasi (2007): *Tree Automata Techniques and Applications*. Available online since 1997: `http://tata.gforge.inria.fr`.

[10] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian & Mohamed Zergaoui (2015): *Early nested word automata for XPath query answering on XML streams*. *Theor. Comput. Sci.* 578, pp. 100–125, doi:10.1016/j.tcs.2015.01.017.

[11] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert & Robert Endre Tarjan (1994): *Dynamic Perfect Hashing: Upper and Lower Bounds*. *SIAM J. Comput.* 23(4), pp. 738–761, doi:10.1137/S0097539791194094.

[12] Massimo Franceschet: *XPathMark Performance Test*. `https://users.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html`. Accessed: 2020-10-25.

[13] Olivier Gauwin, Joachim Niehren & Yves Roos (2008): *Streaming Tree Automata*. *Information Processing Letters* 109(1), pp. 13–17, doi:10.1016/j.ipl.2008.08.002.

[14] Olivier Gauwin, Joachim Niehren & Sophie Tison (2009): *Earliest Query Answering for Deterministic Nested Word Automata*. In: *17th International Symposium on Fundamentals of Computer Theory*, *Lecture Notes in Computer Science* 5699, Springer Verlag, pp. 121–132, doi:10.1007/978-3-642-03409-1_12.

[15] Anthony Lick & Schmitz Sylvain (Last visited April 13th 2022): *XPath Benchmark*. Available at `https://archive.softwareheritage.org/browse/directory/1ea68cf5bb3f9f3f2fe8c7995f1802ebadf17fb5`.

[16] Kurt Mehlhorn (1980): *Pebbling Moutain Ranges and its Application of DCFL-Recognition*. In J. W. de Bakker & Jan van Leeuwen, editors: *Automata, Languages and Programming, 7th Colloquium, No-*

ordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings, Lecture Notes in Computer Science 85, Springer, pp. 422–435, doi:10.1007/3-540-10003-2_89.

[17] Barzan Mozafari, Kai Zeng & Carlo Zaniolo (2012): *High-performance complex event processing over XML streams*. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, Ariel Fuxman, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano & Ariel Fuxman, editors: *SIGMOD Conference*, ACM, pp. 253–264, doi:10.1145/2213836.2213866.

[18] Andreas Neumann & Helmut Seidl (1998): *Locating Matches of Tree Patterns in Forests*. In: *Foundations of Software Technology and Theoretical Computer Science*, *Lecture Notes in Computer Science* 1530, Springer Verlag, pp. 134–145, doi:10.1007/978-3-642-03409-1_12.

[19] Joachim Niehren & Momar Sakho (2021): *Determinization and Minimization of Automata for Nested Words Revisited*. *Algorithms* 14(3), p. 68, doi:10.3390/a14030068.

[20] Joachim Niehren, Momar Sakho & Antonio Al Serhali (2022): *Schema-Based Automata Determinization*. In: *Gandalf*. Available at `https://hal.inria.fr/hal-03536045`.

[21] Alexander Okhotin & Kai Salomaa (2014): *Complexity of input-driven pushdown automata*. *SIGACT News* 45(2), pp. 47–67, doi:10.1145/2636805.2636821.

[22] Markus L. Schmid & Nicole Schweikardt (2021): *A Purely Regular Approach to Non-Regular Core Spanners*. In Ke Yi & Zhewei Wei, editors: *24th International Conference on Database Theory (ICDT 2021)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 186, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 4:1–4:19, doi:10.4230/LIPIcs.ICDT.2021.4.

[23] H. Straubing (1994): *Finite Automata, Formal Logic, and Circuit Complexity*. Progress in Computer Science and Applied Series, Birkhäuser, doi:10.1007/978-1-4612-0289-9.

[24] J. W. Thatcher (1967): *Characterizing derivation trees of context-free grammars through a generalization of automata theory*. *Journal of Computer and System Science* 1, pp. 317–322, doi:10.1016/S0022-0000(67)80022-9.