algorithms

MDPI

Article

Determinization and Minimization of Automata for Nested Words Revisited

Joachim Niehren * and Momar Sakho

Inria, Université de Lille, 59000 Lille, France; momar.sakho@inria.fr
* Correspondence: joachim.niehren@inria.fr

Abstract: We consider the problem of determinizing and minimizing automata for nested words in practice. For this we compile the nested regular expressions (*NREs*) from the usual XPath benchmark to nested word automata (*NWAs*). The determinization of these *NWAs*, however, fails to produce reasonably small automata. In the best case, huge deterministic *NWAs* are produced after few hours, even for relatively small *NREs* of the benchmark. We propose a different approach to the determinization of automata for nested words. For this, we introduce *stepwise hedge automata* (*SHA*s) that generalize naturally on both (stepwise) tree automata and on finite word automata. We then show how to determinize *SHA*s, yielding reasonably small deterministic automata for the *NREs* from the XPath benchmark. The size of deterministic *SHA*s automata can be reduced further by a novel minimization algorithm for a subclass of *SHA*s. In order to understand why the new approach to determinization and minimization works so nicely, we investigate the relationship between *NWAs* and *SHA*s further. Clearly, deterministic *SHA*s can be compiled to deterministic *NWAs* in linear time, and conversely *NWAs* can be compiled to nondeterministic *SHA*s in polynomial time. Therefore, we can use *SHA*s as intermediates for determinizing *NWAs*, while avoiding the huge size increase with the usual determinization algorithm for *NWAs*. Notably, the *NWAs* obtained from the *SHA*s perform bottom-up and left-to-right computations only, but no top-down computations. This *NWA* behavior can be distinguished syntactically by the (weak) single-entry property, suggesting a close relationship between *SHA*s and single-entry *NWAs*. In particular, it turns out that the usual determinization algorithm for *NWAs* behaves well for single-entry *NWAs*, while it quickly explodes without the single-entry property. Furthermore, it is known that the class of deterministic multi-module single-entry *NWAs* enjoys unique minimization. The subclass of deterministic *SHA*s to which our novel minimization algorithm applies is different though, in that we do not impose multiple modules. As further optimizations for reducing the sizes of the constructed *SHA*s, we propose schema-based cleaning and symbolic representations based on apply-else rules that can be maintained by determinization. We implemented the optimizations and report the experimental results for the automata constructed for the XPathMark benchmark.

Keywords: regular expressions; tree automata; nested words; hedges; logical queries; XPath

Citation: Niehren, J.; Sakho, M. Determinization and Minimization of Automata for Nested Words Revisited. *Algorithms* **2021**, *14*, 68. https://doi.org/10.3390/a14030068

Academic Editor: Henning Fernau

Received: 11 December 2020
Accepted: 19 February 2021
Published: 24 February 2021

1. Introduction

Nested words are hierarchical structures that are omnipresent in computer science. They were used to represent sequences of data trees, like XML or JSON documents, and to analyze the call structure of recursive programs. The idea of nested words is to generalize on both words and trees, resulting in sequences of unranked trees that are also known as hedges. Otherwise, nested words can be obtained by enriching Dyck words with internal letters, besides opening and closing parentheses. Furthermore, nested words are the elements of the least set containing internal letters from a given alphabet, triples consisting of an opening parenthesis, a nested word, and a closing parenthesis, and all sequences of nested words. Last but not least, nested words can be seen as words over an alphabet with internal letters, opening parentheses, and closing parentheses, under the conditions that

the parenthesis are well nested, so that every opening parenthesis is properly closed and every closing parenthesis properly opened.

From the viewpoint of formal language theory, a natural question is how to lift the notions of finite automata and regular expressions, from words and trees to nested words, while preserving their well-known relationships. *Nested word automata* (*NWAs*) were heavily studied since the 1980s [1–4], under the name *input-driven automata*. They are the same as *visibly pushdown automata* [5], *pushdown forest automata* [6], and *streaming tree automata* [7]. *NWAs* can recognize the same languages of unranked trees as hedge automata [8], a generalization of tree automata for ranked trees [9]. *NWAs* are often defined as pushdown automata with visible stacks, meaning that exactly one symbol is pushed when reading an opening parenthesis, and exactly one symbol is popped when reading a closing parenthesis, while the stack is not used otherwise. Their main advantage is a powerful notion of determinism, generalizing both over bottom-up and top-down determinism of tree automata for ranked trees [2,3]. We note that general pushdown automata do not permit determinization in contrast.

Regular expressions for nested words were proposed more recently by Hosoya and Pierce [10] under the name of *regular expression types*. In the present article, we will call them *nested regular expressions* (*NREs*) instead. Independently, more complex notions of nested regular expressions were introduced [11,12] in order to deal with generalizations of nested words with dangling opening and closing parentheses, which are not of interest to us. It was already claimed in [10], that our simpler notion of *NREs* has the same expressiveness as hedge automata [8,9], which in turn have the same expressiveness as *NWAs* [3]. However, the question under which conditions *NREs* can be compiled to small deterministic *NWAs* has not been studied. For classes of *NREs* for which deterministic *NWAs* can be computed in polynomial time, we can decide language inclusion or equivalence in polynomial time too. For other classes, these problems may not be feasible since language inclusion for nondeterministic *NWAs* is EXP-complete.

Our concrete interest in the universality of deterministic *NWAs* is motivated by XML stream processing: we want to compute the certain answers of a navigational XPath query on an XML stream [13,14], i.e., those elements that are selected in all possible futures of the stream. Whether an answer is certain is computationally hard, even for tiny syntactic fragments of navigational XPath [14,15], but can be done in polynomial time for queries defined by deterministic *NWAs* [16]. A natural question is, therefore, whether it is possible to compile navigational XPath queries as in the usual benchmark [17] to deterministic *NWAs* of reasonable size. Unfortunately, the existing compilers fail to do so [18], as they are based on *NWA* determinization for dealing with disjunction, negation, and recursive steps. Thereby, they produce huge deterministic automata even for very simple navigational XPath queries from the benchmark, or do not terminate after some hours.

In this article, we consider *NREs* for defining queries on nested words. For benchmarking with realistic example, we consider the navigational XPath queries in the XPathMark benchmark with only forwards axis, that we compiled to *NREs* of the same size up to a constant factor. The question is then whether these *NREs* can be compiled to reasonably small deterministic *NWAs*.

As a first approach, we distinguish a subclass of "deterministic" *NREs* that can be compiled in polynomial time to deterministic *NWAs* by generalizing on Glushkov's construction of deterministic finite-state automata (DFAs) from "deterministic" regular expressions [19,20]. However, the *NREs* obtained by compilation from navigational XPath queries are rarely deterministic, so neither are the *NWAs* compiled from them. Moreover, as we cannot apply *NWA* determinization to them in practice as argued above, this first approach has a much too low coverage to reach the objective. Therefore, we will report it only at the end in Section 9.

For our second approach, we propose a novel variant of automata for nested words that we call *stepwise hedge automata* (*SHA*s). Even though motivated by the wish to create deterministic automata for the *NREs* of our benchmark, they are of general interest:

they generalize naturally on both (stepwise) tree automata [21] and on finite word automata. In contrast to stepwise tree automata, *SHA*s can not only recognize unranked trees, but also sequences thereof, i.e., hedges or nested words. Furthermore, *SHA*s can be determinized in a bottom-up and left-to-right manner by combining in a natural manner the determinization procedures for tree and word automata.

By adapting existing compilers for stepwise tree automata [21], *SHA*s can be compiled to *NWAs* with the same language in linear time while preserving determinism. Conversely, *NWAs* can be compiled to *SHA*s in polynomial time, but at the cost of introducing nondeterminism. By compiling *NWAs* to *SHA*s, determinizing the *SHA*, and compiling the obtained deterministic *SHA* back to a deterministic *NWA*, we can determinize *NWAs* by determinizing the corresponding *SHA*s. This alternative determinization algorithm for *NWAs* is different from the usual determinization algorithm for *NWAs* [2,3,18]. Indeed, it yields reasonably small deterministic *NWAs* for the *NREs* from the XPath benchmark.

Yet another alternative algorithm for determinizing *NWAs* can be obtained by compiling *NWAs* to *SHA*s and back, and then determinizing the *NWAs* obtained in this manner. When applied to back-and-forth converted *NWAs*, the usual *NWA* determinization algorithm turns out to be well behaved: it produces deterministic *NWAs* of reasonable size for all our benchmark *NREs*. This might be surprising, given that the same determinization algorithm behaved so poorly for the non-converted *NWAs* that were obtained from the benchmark *NREs* directly.

We contribute two further solutions for producing deterministic *NWAs* for our benchmark *NREs*. These are both based on a direct compiler from *NREs* to *SHA*s. We can then determinize these *SHA*s, followed by compilation to deterministic *NWAs*. Otherwise, we can first compile the *SHA*s to *NWAs* and then determinize these NWAs.

The next question is why the new determinization algorithms for *NWAs* that use *SHA*s as intermediates work so nicely. In order to understand this, we need to investigate the relationship between *NWAs* and *SHA*s more deeply. Clearly, the *NWAs* obtained via *SHA*s do all their work in a bottom-up and left-to-right manner, and nothing when moving top-down. We can characterize the subclass of *NWAs* with this restricted behavior syntactically by the *(weak) single-entry property*: it requires that all opening rules of the *NWA* go into into the same target state while popping the sources state onto the stack. Note that our single-entry property is weaker than the *(multi-module) single-entry property* studied previously [22–24], which in addition requires that the automaton can be split into at least 2 modules, one for the top level and one for the nested level. The *NWAs* obtained by compilation from *SHA*s all have the (weak) single-entry property (but not necessarily multiple modules). Therefore, when compiling *NWAs* to *SHA*s and back, the resulting *NWAs* also have the (weak) single-entry property. It seems that the usual determinization algorithm from *NWAs* is well-behaved when applied to *NWAs* with the (weak) single-entry property. The relationship between *SHA*s and (weak) single-entry *NWAs* seems sufficiently close, so that their determinization algorithms seem to operate in somehow similar manners.

It is known that the subclass of deterministic multi-module single-entry *NWAs* (also called *call-driven automata*) enjoys unique minimization [22,23]. The separation of the module for the top level from the module for the nested level can be obtained w.l.o.g., by building a product with the *NWA* with two hedge states that distinguishes the two levels. In our application, minimization could thus be used to reduce the size of the deterministic *NWAs* produced by our four algorithms for converting *NREs*, with the hope to eventually obtain a unique outcome after minimization. However, as we will use some symbolic representations for sets of rules, the uniqueness will hold only for the non-symbolic counterpart. In any case, the number of states of the deterministic minimal *NWAs* obtained for the same *NRE* could be expected to become unique.

Motivated by our application, we found it more relevant to minimize deterministic *SHA*s rather than deterministic (weak) single-entry *NWAs* (despite of their close correspondence). As for the class of deterministic (weak) single-entry *NWAs*, a restriction is needed for the class of deterministic *SHA*s to obtain unique minimization. We could have required

the existence of multiple modules as for multi-module single-entry *NWAs*. Instead, we restrict ourselves to deterministic *SHA*s for which the initial states for trees and hedges coincide. We then show that minimization for such *SHA*s can be reduced to the minimization of tree automata up to a novel encoding of hedges to binary trees.

We implemented all four algorithms for compiling our benchmark *NREs* to deterministic *NWAs* and report the experimental results. We have also implemented the novel minimization algorithm for *SHA*s with equal tree and hedge initial states, and used it in our experiments. We propose two further optimization methods for reducing the sizes of the constructed automata.

First, we introduce schema-based cleaning both for *SHA*s and *NWAs*. In our application, the schema expresses the XML data model, stating that hedges must encode valid XML documents. More generally, an automaton A can be cleaned relative to an automaton S for the schema, if the language of interest is the intersection $L(A) \cap L(S)$ rather than $L(A)$ itself. The idea of schema-based cleaning is to keep only those transition rules of A that are used to recognize some hedge of $L(S)$. These transition rules can be computed from the product of A and S. Note that schema-based cleaning may change the language of the automaton. Only the intersection $L(A) \cap L(S)$ is preserved, not necessarily $L(A)$.

Second, we propose a symbolic representations for *SHA*s based on apply-else rules. They help to represent more compactly a large number of apply rules produced by the determinization of *SHA*s. Before compiling *SHA*s to *NWAs*, however, we need to eliminate the apply-else rules. This is because we have not developed analogous symbolic representations for *NWAs* so far. A second limitation is that we have not implemented any minimization algorithm for *NWAs* at the time being.

The main improvement of this journal article compared with the conference version [25] is the addition of the minimization algorithm for the subclass of *SHA*s with equal tree and hedge initial states. Furthermore, we added the idea of schema-based cleaning and the symbolic representations for *SHA*s by apply-else rules. The experimental results were enhanced with minimization, symbolic representations of rules, and schema-based cleaning. All the nested regular expressions generated for the XPathMark benchmark queries that we consider, as well as their corresponding automata—when we could produce them—can be found at http://researchers.lille.inria.fr/niehren/complementary-material (accessed on 1 February 2021).

Related Work. In the present article, we restrict ourselves to nested words over signatures with a single opening parenthesis, a single closing parenthesis, and possibly many internal letters a, b. This permits us to simplify the presentation of nested regular expressions, the notions of *NWAs* and *SHA*s, their forth and back compilers, as well as the determinization algorithms. Note that any multi-module *NWA* for such signatures must have exactly 2 modules. From an application perspective, multiple parentheses can be encoded by using internal letters, that is a named opening parenthesis $\langle_a$ by the word $a \cdot \langle$ and a named closing parenthesis $\rangle_b$ by the word $b \cdot \rangle$. When encoding XML documents as nested words, such some encoding is needed anyway in order to deal with the complex information in XML tags, and also to provide symbolic representations with else rules that are able to deal with infinite signatures.

For the minimization of deterministic *NWAs*, general signatures with multiple parentheses raise additional problems. Chervet and Walukiewicz [23] solved such problems by reducing the minimization for *expanded CDAs* to the minimization of *CDAs*. Gauwin, Muscholl, and Raskin [24] showed that the minimization for deterministic *NWAs* is NP-hard in the case with general signatures. Their approach is based on a reduction from the problem of minimal immersion for sequences of DFAs, for which they construct *NWAs* with an unbounded number of opening parenthesis and an unbounded number of entry states. Weak single-entry *NWAs* in our setting do not permit this. Neither do *NWAs* over fixed general signatures with a finite number of opening parenthesis.

Navigational XPath queries on XML documents can be formalized in the language CoreXPath [26], or more generally by nested regular path queries [27] on data trees. Nested regular path queries were introduced earlier under the name of the propositional dynamic logic (PDL) in the 1970s [28], where they are applied to labeled graphs that generalize on data trees.

As certain query answering for XPath was considered difficult, the currently existing approaches to XPath query evaluation on XML streams [13,18] either approximate certain query answers based on nondeterministic machines or restrict the queries so that answer certainty can be decided without latency [15,29]. This also holds for recent streaming algorithms on words without nesting in the context of complex event processing [30].

2. Nested Words

Nested words are words with parentheses that are well nested. They can be identified with hedges, that is, sequences of internal symbols and unranked trees.

Nested words are constructed with opening and closing parentheses, respectively, $\langle$ and $\rangle$. An unranked alphabet Σ is a possibly infinite set of so-called "internal" symbols, that does not contain the two parentheses. The set of nested words over Σ is denoted $\mathcal{N}_\Sigma$ and is defined by the following abstract syntax:

$$h, h' \in \mathcal{N}_\Sigma \quad ::= \quad \varepsilon \quad | \quad a \quad | \quad \langle h \rangle \quad | \quad h \cdot h' \qquad \text{where } a \in \Sigma$$

The empty nested word is denoted by ε and assumed to be the neutral element of the composition operator $\varepsilon \cdot h = h = h \cdot \varepsilon$, which furthermore is assumed to be associative, i.e., $h_1 \cdot (h_2 \cdot h_3) = (h_1 \cdot h_2) \cdot h_3$.

Nested words can be identified with hedges, i.e., words of trees and internal symbols. Seen as a graph, the inner nodes are labeled by the tree constructor $\langle\rangle$ and the leaves 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 in Figure 1. A nested word of type *tree* has the form $\langle h \rangle$.

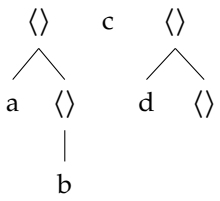

Figure 1. Nested word $\langle a \cdot \langle b \rangle \cdot \varepsilon \rangle \cdot c \cdot \langle d \cdot \langle \varepsilon \rangle \rangle$ seen as a graph.

Variants. Our notion of nested words accepts only well-nested words without dangling opening or closing parentheses in contrast to others [3,5]. This will lead to simpler notion of regular expressions, avoiding the more complex operators as with visibly rational expressions [12,31]. A less important difference is that we do not support labeled parentheses.

Labeled unranked trees. Labeled parentheses can be simulated by using internal letters. For instance, the labeled tree $a(b(), c())$ can be represented by the nested word of type tree $\langle a \cdot \langle b \rangle \cdot \langle c \rangle \rangle$. In this way, the labeled tree $a()$ is represented by the nested word $\langle a \rangle$, which is of type tree (while the internal letter a alone is not). Unranked sequences of subtrees, often called hedges and sometimes forests, can be composed by using the sequence operator.

XML Documents. Our notion of nested words is sufficiently powerful to express general XML documents. An example of an XML document is given in Figure 2 and the representing nested word in Figure 3.

```
⟨site⟩
  ⟨closed_auctions⟩
    ⟨closed_auction⟩
      ⟨date⟩01/01/2000⟨/date⟩
      ⟨keyword⟩wine⟨keyword⟩
    ⟨/closed_auction⟩
  ⟨/closed_auctions⟩
⟨/site⟩
```

Figure 2. An XML document.

```
⟨doc·
  ⟨elem · site·
    ⟨elem · closed_auctions·
      ⟨elem · closed_auction·
        ⟨elem · date · 0 · 1 · / · 0 · 1 · / · 2 · 0 · 0 · 0⟩
        ⟨elem · keyword · w · i · n · e · ⟩
      ⟩
    ⟩
  ⟩
⟩
```

Figure 3. The corresponding nested word.

We use the names of XML elements as labels of the nested word, as well as the letters of UTF8 for the string data values. Further labels such as *doc* and *elem* are added to express the types of the XML data model document and element, respectively.

When it comes to querying for nodes in XML documents, we will be interested in nested words encoding XML documents, in which a unique node is marked. We will use the label x to mark the selected node and the label $\neg x$ for all others. When marking the *date* in the XML document of Figure 2, we obtain the nested word in Figure 4.

```
⟨doc · ¬x·
  ⟨elem · site · ¬x·
    ⟨elem · closed_auctions · ¬x·
      ⟨elem · closed_auction · ¬x·
        ⟨elem · date · x · 0 · 1 · / · 0 · 1 · / · 2 · 0 · 0 · 0⟩
        ⟨elem · keyword · ¬x · w · i · n · e · ⟩
      ⟩
    ⟩
  ⟩
⟩
```

Figure 4. The nested word of the x-marked XML document from Figure 2.

3. Nested Regular Expressions

We present nested regular expressions (*NREs*), which were introduced under the name *regular expression types* in the context of XDuce [10] up to minor details. Note that similar nested regular expressions for ranked trees are folklore in the context of tree automata [32].

3.1. Syntax and Semantics

Let the alphabet Σ be a set. An *NRE* over Σ is a term describing a language of nested words. It has the following abstract syntax where $a \in \Sigma$:

$$E, E' ::= \varepsilon \mid a \mid \_ \mid \emptyset \mid E \cdot E' \mid E + E' \mid E \& E' \mid E^* \mid \overline{E} \mid \langle E \rangle \mid \mu a.E$$

The $\mu a.E$ expressions are the same as in μ-calculus [33], except that we restrict them such that all occurrences of a in E are nested below parentheses. Otherwise, nonregular

languages could be defined such as with $\mu a.\ (b \cdot a \cdot c + \varepsilon)$ whose language would be $\{b^n \cdot c^n \mid n \in \mathbb{N}\}$. We also forbid intersections and complements in expression $\mu a.E$ on all paths between the μa-operator and the occurrences of a in E that are bound by this operator. The expressions $\mu a.E$ allow for vertical recursion, while the expressions with the Kleene star E^* support horizontal recursion.

Our syntax allows for conjunctions $E\&E'$ and negations $\overline{E}$, which are well known to not add expressiveness if Σ is finite. They are still relevant from the viewpoints of modeling, and for the treatment of infinite signatures. This comes at the price of increasing the complexity, as for the well-known case of words [34].

For infinite signatures, we can define for any finite subset Σ' of labels the language of single-letter words $\Sigma \setminus \Sigma'$ by some *NRE*. This can be seen as follows. If $\Sigma' = \emptyset$, then the expression _ does the job: it matches exactly the set of all labels in Σ. Moreover, if Σ' is nonempty then we can use negation. For instance, if $\Sigma' = \{a, b\}$ then the expression $\overline{a + b}\&\_$ describes the language $\Sigma \setminus \Sigma'$.

The sets of free and bound letters $fn(E)$ and $bn(E)$ are defined as usual. The only binder $\mu a.E$ binds the symbol a with scope E. Note that $fn(\_) = \emptyset$.

There are three differences with respect to the regular expression types from [10]. First, our *NREs* treat labels as internal symbols instead of labels of parentheses. Second, they provide recursion through the μ-operator instead of using recursive equation systems. Third, conjunctions and general negations are not considered there.

Any *NRE* E describes a language $L(E)$ of nested words that we define by induction on the structure of E as follows: $\mathcal{N}_\Sigma$ is the set of nested words over Σ, as defined in Section 2.

$$\begin{array}{ll}
L(\varepsilon) = \{\varepsilon\} \qquad L(a) = \{a\} & L(_) = \Sigma \\
L(E \cdot E') = L(E) \cdot L(E') & L(\overline{E}) = \mathcal{N}_\Sigma \setminus L(E) \\
L(E + E') = L(E) \cup L(E') & L(E\&E') = L(E) \cap L(E') \\
L(\langle E \rangle) = \{\langle h \rangle \mid h \in L(E)\} & L(\mu a.E) = \cup_{n \geq 0} L(\mu^n a.E) \\
L(E^*) = L(E)^* & L(\emptyset) = \emptyset
\end{array}$$

For all expressions—E, E_1, and E_2, the notation $E[E_1/E_2]$ stands for the expression E where all the occurrences of E_1 have been replaced by E_2. The semantics of a μ-operator is then defined using the shortcuts $\mu^0 a.E = E[a/\emptyset]$ and $\mu^n a.E = E[a/\mu^{n-1} a.E]$ for all $n \geq 1$. In particular $L(\mu a.\_) = L(\_) = \Sigma$, so that $a \in L(\mu a.\_)$. The semantics of the complement expression $L(\overline{E})$ is the complement of $L(E)$ in the set of all nested words, that is $\mathcal{N}_\Sigma \setminus L(E)$.

3.2. XPATH *Example*

We now show how to express navigational XPATH queries by *NREs* that are restricted to forward axis. The idea is to adapt the spirit of a generate-and-test algorithm for query answering. The generation produces a nested word from XML documents by guessing a single node and marking it by x. This node is a candidate for a query answer that is to be tested. The test is done by a *NRE*.

For expressing XPATH queries with `child` and `descendant-or-self` axes we will use the following *NREs* where $a \notin fn(E)$:

$$\begin{array}{lll}
T & =_{\text{df}} & \mu a.\ (\langle a \rangle + _)^* \\
ch(E) & =_{\text{df}} & T \cdot \langle E \rangle \cdot T \\
ch^*(E) & =_{\text{df}} & \mu a.\ (E + ch(a)) \\
ch^+(E) & =_{\text{df}} & \mu a.\ (ch(E) + ch(a))
\end{array}$$

For instance, consider the XPATH query A5 from the XPathMark benchmark [17]:

```
/site/closed_auctions/closed_auction[descendant::keyword]/date
```

Applied to the above XML document, it selects all `date` children of `closed_auctions` nodes that contain at least one `keyword` descendant. Query A5 can be compiled to the following *NRE*, which will accept the nested word in Figure 4 in particular:

$$\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot ch(elem \cdot closed\_auctions \cdot \_ \cdot ch(\\ elem \cdot closed\_auction \cdot \_ \cdot (ch^+(elem \cdot keyword \cdot \_ \cdot T) \,\&\, ch(elem \cdot date \cdot x \cdot T)))))\rangle\rangle$$

The only label that the expression _ may match on a document that is properly annotated with the variable x will be the letter $\neg x \in \Sigma$. The label x is annotated to the marked node, which is tested for being selected by the query. The label $\neg x$ is annotated to all nodes except a unique x-marked node.

Note also that the μ-operator of the $ch^+(\ldots)$-expression expresses the recursion of the descendant axis. Furthermore, the conjunction permits us to connect the main path of A5 with its only filter.

3.3. XPath Benchmark

For testing *NREs*, we rely on the usual XPathMark benchmark [17]. We restrict ourselves to navigational path queries with forward axis: `child`, `descendant`, and `following-sibling`. We notice that the `following` axis is excluded in contrast to `following-sibling`, as `following` is not strictly forwards. We can also admit path composition and filters with conjunction, disjunction, and negation.

The XPATH queries of the benchmark satisfying these restrictions are the queries A1, ..., A8 and B3 given in Figure 5. We developed a more general compiler from navigational forward XPATH queries to *NREs*, which yields the *NREs* in Figure A1 of the Appendix B for the benchmark XPATH queries. The *NREs* for A1–A3 do have neither conjunctions nor negations, while the queries A4–A8 contain filters, which are mapped to conjunctions in *NREs*. The compiler uses the μ-operator to capture the recursion of `descendant` axis as in A2, A3, and A5. Furthermore, nondeterminism is introduced by disjunctions in filters as in A7 and A8. Conjunction in filters appears in A6 which is mapped to conjunctions in *NREs* too. A detailed description of this compiler is not in the scope of the present article though.

```
A1: /site/closed_auctions/closed_auction/annotation/description/text
    /keyword
A2: //closed_auction//keyword
A3: /site/closed_auctions/closed_auction//keyword
A4: /site/closed_auctions/closed_auction
    [annotation/description/text/keyword]/date
A5: /site/closed_auctions/closed_auction[descendant::keyword]/date
A6: /site/people/person[profile/gender and profile/age]/name
A7: /site/people/person[phone or homepage]/name
A8: /site/people/person
    [address and (phone or homepage) and (creditcard or profile)]
    /name
B3: /site/open_auctions/open_auction/bidder[following-sibling::bidder]
```

Figure 5. XPath benchmark queries.

4. Nested Word Automata

Nested word automata (*NWAs*) are pushdown automata reading nested words, whose stacks are visible: they push a single stack symbol when reading an opening parenthesis, pop a single stack symbol when reading a closing parenthesis, and do not alter or inspect the stack otherwise.

Definition 1. *An NWA is a tuple* $A = (Q_h, Q_t, \Sigma, \Gamma, \Delta, I, F)$ *consisting of a possibly infinite set* Σ *of internal symbols; finite sets* Q_h *and* Q_t *of states of type hedge and tree, respectively; sets of*

initial and final states $I, F \subseteq Q_h$*; a finite set* Γ *of stack symbols; and a finite set* Δ *of transition rules of the forms:*

hedge rules	$a^\Delta, \_^\Delta, \varepsilon^\Delta \subseteq Q_h \times Q_h$	*where* $a \in \Sigma$
opening rules	$\langle^\Delta_\gamma \subseteq Q_h \times Q_h$	*where* $\gamma \in \Gamma$
tree rules	$\mathrm{T}^\Delta \subseteq Q_h \times Q_t$	
closing rules	$\rangle^\Delta_\gamma \subseteq Q_t \times Q_h$	

Our *NWAs* are symbolic, in that they come with else rules, i.e., elements of $(q, q') \in \_^\Delta$ that we will denote by $q \Rightarrow q'$, for dealing with large or infinite alphabets.

An example for an *NWA* is given in a graphical syntax in Figure 6. Tree states are drawn in circles that are filled in light gray (q), while hedge states are in unfilled circles (q). Initial states are drawn as →(q) and final states as ((q)). Hedge rules that have the form $(q_1, q_2) \in o^\Delta$ where $o \in \Sigma \cup \{\_, \varepsilon\}$ are denoted by $q_1 \xrightarrow{o} q_2$, while any tree rule $(q_1, q_2) \in \mathrm{T}^\Delta$ is denoted $q_1 \dashrightarrow q_2$. Opening rules $(q_1, q_2) \in \langle^\Delta_\gamma$ are represented as $q_1 \xrightarrow{\downarrow\gamma} q_2$ and closing rules $(q_1, q_2) \in \rangle^\Delta_\gamma$ as $q_1 \xrightarrow{\uparrow\gamma} q_2$.

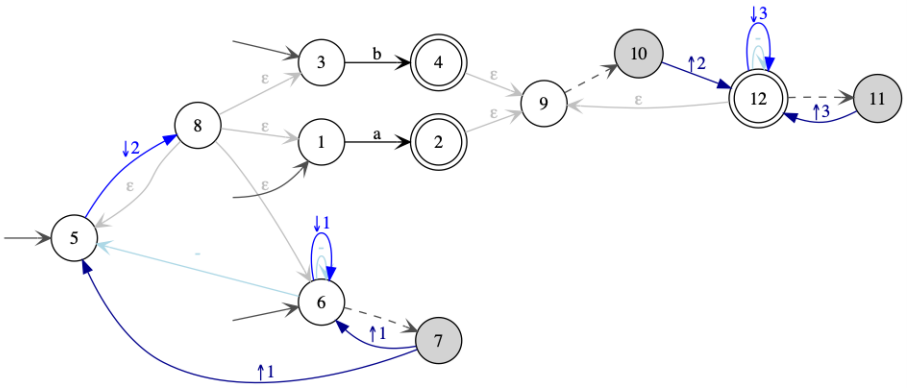

Figure 6. Nested word automaton $nwa(ch^*(a + b))$.

Our notion of *NWAs* supports factorization in the spirit of the work in [35]. It is obtained by distinguishing two types of states, $q \in Q_h$ and $p \in Q_t$, and adding explicit type coercion rules $q \dashrightarrow p$. Semantically, both kinds of states could be merged when replacing the type coercion rules by the epsilon rule $q \xrightarrow{\varepsilon} p$, but at the cost of introducing additional nondeterminism. This may lead to quadratically larger deterministic automata, as we will illustrate at the *NWA* in Figure 20.

The language of nested words between two states $q_1, q_2 \in Q_h$ is defined as the least language such that

$$\begin{aligned} L_{q_1,q_2}(\Delta) \quad = \quad & \{\varepsilon \mid \text{if } q_1 = q_2 \text{ or } q_1 \xrightarrow{\varepsilon} q_2 \in \Delta\} \cup \bigcup_{q_3 \in Q_h} L_{q_1,q_3}(\Delta) \cdot L_{q_3,q_2}(\Delta) \\ \cup \quad & \{a \mid \text{if } q_1 \xrightarrow{a} q_2 \in \Delta \text{ or } (q_1 \Rightarrow q_2 \in \Delta \text{ and } \neg \exists q_2'.\ q_1 \xrightarrow{a} q_2' \in \Delta)\} \\ \cup \quad & \{\langle h \rangle \mid \exists q_1', q_2' \in Q_h . \exists q_3 \in Q_t . \exists \gamma \in \Gamma.\ q_1 \xrightarrow{\downarrow\gamma} q_1', h \in L_{q_1',q_2'}(\Delta), \\ & \qquad q_2' \dashrightarrow q_3 \in \Delta \text{ and } q_3 \xrightarrow{\uparrow\gamma} q_2 \in \Delta\}. \end{aligned}$$

The language of the *NWA* is then $L(A) = \bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(\Delta)$.

4.1. Determinization of NWAs

Determinization for *NWAs* was first studied by von Braunmühl and Verbeek [2] in the 1980s, where *NWAs* are named *input-driven pushdown automata*. We notice that the determinization algorithm was published only in the journal version of this paper, but not

in the conference version. Later on, the same algorithm was rediscovered in the context of visibly pushdown automata and republished for nested word automata.

Definition 2. *An NWA A is called deterministic or equivalently a dNWA if*

- *I contains at most one element;*
- *there is no epsilon rule, i.e., $\varepsilon^\Delta = \emptyset$,*
- *a^Δ and $\_^\Delta$ are partial functions from Q_h to Q_h for all $a \in \Sigma$, and T^Δ is a partial function from Q_h to Q_t;*
- *for all $q \in Q_h$ and $\gamma \in \Gamma$ there exists a most one $q' \in Q_h$ such that $q' \in \langle^\Delta_\gamma$; and*
- *$\rangle^\Delta_\gamma$ is a partial function from Q_t to Q_h for all $\gamma \in \Gamma$.*

Proposition 1 (von Braunmühl and Verbeek [2])**.** *A NWA with n states can be determinized in time $O(2^{n^2})$.*

Many of our results are based on the determinization algorithm going back to von Braunmühl and Verbeek. For self-containedness, we recall the version of this algorithm that we will use in the Appendix A. For illustrations, the determinization of the *NWA* in Figure 6 is also presented here too, see Figure A2. It has size 271 while the nondeterministic *NWA* has size 39 (12 states + 2 letters + 3 stack symbols + 22 rules). The blow-up is even worse in general as our experimental results will show and as noticed earlier by [18].

4.2. Multi-Module NWAs

Multi-module *NWAs* will play a prominent role for our *NWA* constructions and are relevant for minimization [23]. For signatures with a single opening parenthesis, each multi-module *NWAs* has exactly two modules, one for the top level and one for the nested level.

We can define multi-module *NWAs* based on the natural notion of homomorphisms for *NWAs*. A homomorphism from an *NWA* A to an *NWA* A' with the same signature is a triple of functions $(\alpha_h \colon Q_h \to Q'_h, \alpha_t \colon Q_t \to Q'_t, \beta : \Gamma \to \Gamma')$ that maps all concepts of A to the corresponding concepts of A'. These concepts are hedge initial states, final states, opening, closing, internal, and tree transitions. We do not enforce the preservation of epsilon rules by homomorphisms.

Definition 3. *A* multi-module *NWA A is an NWA for which there exists a homomorphism from A to the NWA $\mathcal{T}$ in Figure 7.*

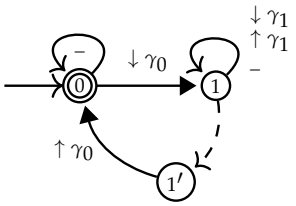

Figure 7. The *NWA* $\mathcal{T}$ maps top level positions to state 0 and nested positions to 1 or $1'$.

The *NWA* $\mathcal{T}$ evaluates all top level positions of a nested word to state 0: all those positions that are not between parentheses. All nested positions are evaluated to state 1. The homomorphism of a multi-module *NWA* A to $\mathcal{T}$ thus partitions the states of A between those that can be assigned to top level positions, and the others that can be assigned to nested positions.

4.3. Compilation of NREs to NWAs

We next discuss a compiler from *NREs* E to *NWAs* $nwa(E)$. This compiler extends on the McNaughton–Yamada–Thompson algorithm [36] for regular expressions, which introduces epsilon edges for constructing the automata of composition $E \cdot E'$.

Theorem 1. *For any NRE E, we can construct an NWA A such that $L(A) = L(E)$. If E contains neither conjunctions nor negations, then the construction is in time $O(|E|)$.*

Proof sketch. Conjunctions $E\&E'$ are compiled to products of automata, so repeated conjunctions may lead to an exponential blow up. Negations $\overline{E}$ are computed by complementing automata based on determinization. Each complementation may lead to an exponential blow-up, so when this is repeated, the construction may become non-elementary.

For expressions without conjunction and negation, no such blow-up may arise. As stated by the theorem, we have to show that expressions can be compiled in linear time.

Case $E = E' \cdot E''$**:** We use the McNaughton–Yamada–Thompson algorithm for composing the *NWAs* of $nwa(E')$ and $nwa(E'')$.

Case $E = \langle E' \rangle$**:** Let Q'_h, Q'_t, and Γ' be, respectively, the set of hedge states, tree states, and stack symbols of $nwa(E')$. We consider new hedge states q_i and q_f that are not in Q'_h, a new tree state p not in Q'_t and a new stack symbol γ not in Γ'. Then, $nwa(E)$ is constructed by adding to $nwa(E')$ opening rules $q_i \xrightarrow{\downarrow\gamma} q$ for all the initial states q of $nwa(E')$, tree rules $q' \longrightarrow p$ for all the final states q' of $nwa(E')$ and a closing rule $p \xrightarrow{\uparrow\gamma} q_f$. Furthermore, we set q_i as the only initial state of $nwa(E)$, and q_f as its sole final state.

Case $E = \mu a.E'$**:** Special care has to be given to repeat expression $\mu a.E$. First of all, the naive compilation approach for these expression turns out to be wrong. Second, fixing the problem in the simplest possible manner does not lead to a linear time algorithm.

Note that we can assume w.l.o.g. that a occurs at most once in E by using the golden lemma of the μ calculus [37], stating for all names $a_1, \ldots, a_n$ and expressions E'' in which $a_1, \ldots, a_n$ can appear free that $\mu a_1 \ldots . \mu a_n.\, E'' \equiv \mu a.E''[a_1/a, \ldots, a_n/a]$. Our construction guarantees that all transitions of the form $q \xrightarrow{a} q'$ in $nwa(E)$ will start with the same state q. The wrong naive construction would remove the transitions $q \xrightarrow{a} q'$ from $nwa(E)$ and add ε-rules from q to all the initial states of $nwa(E)$, and from all final states of $nwa(E)$ to q'. Unfortunately, the construction is not correct. For illustration, we consider the *NRE* $E = \mu a.\langle a^* \rangle$. The reader should be warned that constructing an *NWA* for E is less trivial than it might seem at first sight. One has to start from the *NWA* for $\langle a^* \rangle$ which is given in Figure 8. Simply adding epsilon edges to capture the operator μa will not work though. It will lead to the wrong automaton in Figure 9. This automaton will wrongly accept the hedge $\langle\rangle\langle\rangle$, as this hedge does not belong to $L(E)$.

If the *NWA* for E is multi-module, then the naive construction of compiling $\mu a.E$ can be made correct. Therefore, the simplest fix is to make the *NWA* multi-moduled, before applying the naive construction. This can be achieved by *typing* the states of the automaton, by states of the *NWA* $\mathcal{T}$ in Figure 7. The added types yield the homomorphism of the constructed automaton to $\mathcal{T}$.

The naive algorithm is then adapted as follows. Let $\mathcal{P}$ be the multi-module *NWA* obtained from the product of $nwa(E)$ and $\mathcal{T}$. Note that we keep only the accessible top level states (type 0), but all nested states (type 1). In our example, this yields the *NWA* in Figure 10. We then remove transition $(q, 1) \xrightarrow{a} (q', 1)$ and add ε-rules from state $(q, 1)$ to all states in $I \times \{1\}$, and from all states in $F \times \{1\}$ to $(q', 2)$, where I and F are, respectively, the set of initial and final states of $nwa(E)$. Then, $\mathcal{P}$ recognizes $L(\mu a.E)$. The result obtained in the example is shown in Figure 11.

The algorithm described so far makes the *NWA* multi-moduled before compiling a μ-operator. For this, two copies of all states are introduced. This, however, could lead to an exponential construction if multiple μ-operators are nested. This problem can be avoided by preserving multi-moduledness as an invariant. Whenever a new state is created, it is created twice: once for the top level and once for the nested level.

This information is maintained by typing the states, so that no further copies of the same state are produced later on.

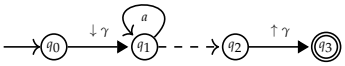

Figure 8. Automaton for the $\langle a^* \rangle$ expression.

Figure 9. Wrong naive construction for $\mu a.\langle a^* \rangle$.

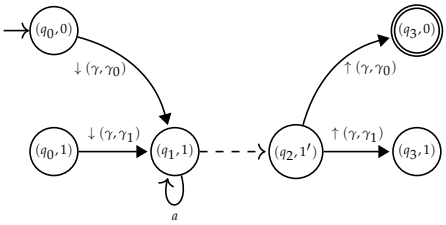

Figure 10. The multi-module *NWA* for $\langle a^* \rangle$.

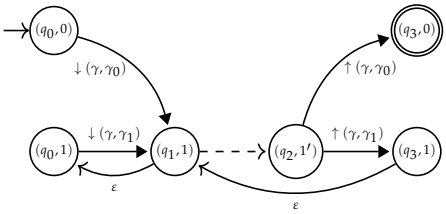

Figure 11. The correctly adapted naive construction for $\mu a.\langle a^* \rangle$.

We omit the correctness proof of this construction. □

4.4. Experimental Results Starting with the NWA Compiler

In the first two column of Figure 12, we report the sizes of the *NWAs* obtained from *NREs* by our compiler, and the size of the deterministic *NWAs* produced thereof. For each automaton, we give its total size and in parentheses the number of states.

	nwa(.)	det(nwa(.))	step(nwa(.))	det(nwa(step(nwa(.))))	nwa(det(step(nwa(.))))
A1	221 (68)	—	231 (88)	398 (37)	409 (37)
A2	185 (49)	362,600 (6782)	224 (81)	4105 (148)	1659 (127)
A3	189 (54)	318,704 (8216)	213 (79)	907 (62)	635 (56)
A4	625 (193)	—	414 (159)	487 (42)	499 (42
A5	486 (135)	—	516 (190)	1192 (73)	868 (67)
A6	2170 (653)	—	1005 (391)	548 (45)	561 (45)
A7	434 (135)	—	378 (146)	468 (41)	480 (41)
A8	10,597 (3127)	—	21,848 (7022)	—	—
B3	253 (77)	—	239 (91)	423 (38)	407 (37)

Figure 12. The size (#states) of the nested word automata (*NWAs*) for the benchmark nested regular expressions (*NREs*) and the automata derived thereof.

The sizes of the nondeterministic *NWAs* produced by the compiler for the *NREs* for A1–A8 and B3 are given in column $nwa(.)$ of Figure 12. Note that the *NWAs* are cleaned so that only accessible and co-accessible states remain. The sizes of the nondeterministic *NWAs* are acceptable for all *NREs*, except for A8, for which the *NWA* has more than 3000 states and an overall size greater than $10,000$. This can be partially explained by the fact that

the *NRE* for A8 contains three conjunctions (one for the filter and two for the conjunctions in the filter). Still, the number of states remains surprising.

The determinized *NWAs* are given in column $det(nwa(.))$. It turns out that only A2 and A3 could be determinized successfully with some few hours of computation time on a standard laptop. However, even in the successful cases, the resulting deterministic *NWAs* are simply huge. This confirms similar problems first noticed in [18] and not solved since then.

The remaining columns of Figure 12 based on the back-and-forth compiler from *SHAs* to *NWAs* from the following Section 6. They show that better determinization algorithms can indeed be obtained, yielding *NWAs* of acceptable size for all benchmark queries, with the exception of A8. The idea of $det(nwa(step(nwa(.))))$ is to compile the *NWAs* obtained from the *NREs* to stepwise hedge automata and back before applying the above algorithm for *NWAs*. This might be surprising, as this determinization algorithm failed for the original *NWAs*, while it now proves successful on the forth-and-back transformed *NWAs*.

5. Stepwise Hedge Automata

We propose *SHAs* as an extension of stepwise tree automata [21] that allows to recognize not only unranked trees but also hedges. We avoid more classical hedge automata from [9] that were already introduced in 1967 by Thatcher [8], as their notion of determinism is problematic. For instance, it makes unique minimization fail [38] and universality hard.

Our notion of *SHAs* will be symbolic in using else rules, and factorized as in [35]: there are two types of states for hedges and trees and an operator for explicit type coercion. We also propose a novel treatment of internal letters inspired by nested word automata, so that *SHAs* generalize both on stepwise tree automata and on NFAs.

Definition 4. *A SHA is a tuple* $A = (Q_h, Q_t, \Sigma, \Delta, I, F)$ *such that* Q_t *and* Q_h *are finite sets of states of two types: t for tree and h for hedge, respectively;* Σ *an alphabet of internal letters (that may be infinite);* $I, F \subseteq Q_h$ *are subsets of hedge initial and final states, respectively; and* Δ *is a finite set of transition rules such that for all* $q \in Q_t$ *and* $a \in \Sigma$*:*

$$\begin{array}{ll} \textit{hedge rules} & q^{\Delta}, a^{\Delta}, \_^{\Delta}, \varepsilon^{\Delta} \subseteq Q_h \times Q_h \\ \textit{tree final rules} & T^{\Delta} \subseteq Q_h \times Q_t \\ \textit{tree initial states} & \langle\rangle^{\Delta} \subseteq Q_h \end{array}$$

An example for a *SHA* is given in graphical syntax in Figure 13. It recognizes all hedges that are either just a or b or contain some tree node that contains either just a or b. In the graphical syntax, the states of type tree $q \in Q_t$ are drawn in circles filled in light gray (q), while the states of type hedge $q' \in Q_h$ are drawn in unfilled circles (q'). The right part of the graph is an NFA which uses tree states as additional edge labels, while the left part is a stepwise tree automaton that defines the tree languages of these tree states.

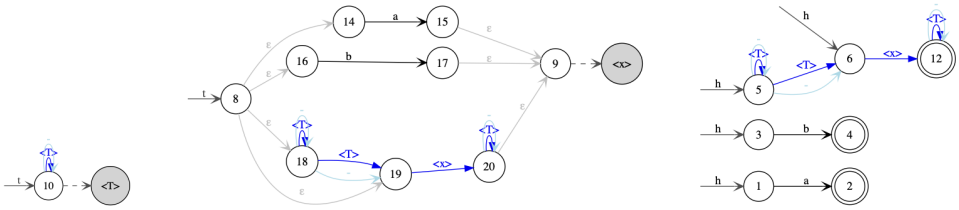

Figure 13. Stepwise hedge automaton $\text{step}(ch^*(a+b))$: the part with the stepwise tree automaton is on the left and middle, and the NFA part on the right.

Let Δ_h be the restriction of Δ to the hedge rules. Then, $(Q_h, \Sigma \uplus Q_t, \Delta_h, I, F)$ is a standard NFA with ε-rules, which is symbolic [39] in providing else rules for dealing with large or infinite alphabets in addition. Therefore, we denote the hedge initial states $q \in I$

by $\xrightarrow{h}(q)$ and the final states $q \in F$ by $((q))$. A rule with an internal letter $(q_1, q_2) \in a^\Delta$ is denoted by $q_1 \xrightarrow{a} q_2 \in \Delta$ stating that a hedge in state q_1 can be extended by the internal letter a leading to a hedge in state q_2. Similarly, an epsilon rule $(q_1, q_2) \in \varepsilon^\Delta$ is denoted by $q_1 \xrightarrow{\varepsilon} q_2$, and an else rule $(q_1, q_2) \in \_^\Delta$ is denoted by $q_1 \Rightarrow q_2$. In the same spirit, a hedge rule $(q_1, q_2) \in q^\Delta$—also called apply rule—is denoted by $q_1 \xrightarrow{q} q_2 \in \Delta$, stating that a hedge in state q_1 can be extended by a tree in state q leading to a hedge in state q_2.

A tree initial state $q \in \langle\rangle^\Delta$ is graphically denoted by $\xrightarrow{t}(q)$ and a tree final rule $(q_1, q_2) \in \mathrm{T}^\Delta$ by $q_1 \longrightarrow q_2$. Intuitively, a tree $\langle h \rangle$ can be evaluated to state q if h can be evaluated starting with some tree initial state $q_1 \in \langle\rangle^\Delta$ to some state q_2 such that $q_2 \longrightarrow q \in \Delta$. More formally, the hedge languages $L_{q_1,q_2}(A)$ between any two hedge states $q_1, q_2 \in Q_h$ are defined as

$$\begin{array}{rcl} L_{q_1,q_2}(A) & = & \{\varepsilon \mid \text{if } q_1 = q_2 \text{ or } q_1 \xrightarrow{\varepsilon} q_2 \in \Delta\} \cup \bigcup\limits_{q_3 \in Q_h} L_{q_1,q_3}(A) \cdot L_{q_3,q_2}(A) \\ & \cup & \{a \mid \text{if } q_1 \xrightarrow{a} q_2 \in \Delta \text{ or } (q_1 \Rightarrow q_2 \in \Delta \text{ and } \neg \exists q_2'.\ q_1 \xrightarrow{a} q_2' \in \Delta)\} \\ & \cup & \bigcup\limits_{q_1 \xrightarrow{q} q_2 \in \Delta} L_q(A) \end{array}$$

This definition is mutually recursive with the definition of the tree languages $L_q(A)$ of all tree states $q \in Q_t$:

$$L_q(A) \quad = \quad \{\langle h \rangle \mid \xrightarrow{t}(q_1) \in \Delta,\ h \in L_{q_1,q_2}(A),\ q_2 \longrightarrow q \in \Delta\}$$

The hedge language $L(A)$ that is recognized by the automaton is $\bigcup\limits_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(S)$. The rules of standard bottom-up tree automata have the form $a(q_1, \ldots, q_n) \to q$ where a is a symbol of arity n. With *SHA*s, this rule can be encoded by the sequence $\xrightarrow{t}(p_0) \xrightarrow{a} (p_1) \xrightarrow{q_1} \ldots \xrightarrow{q_n} (p_n) \longrightarrow q$ where the states $q_1, \ldots, q_n, q$ are all tree states, and $p_0, \ldots, p_n$ new hedge states.

5.1. Determinization of SHAs

We formalize the notion of determinism for stepwise hedge automata and show how determinization works.

Definition 5. *A SHA $(Q_h, Q_t, \Sigma, \Delta, I, F)$ is* deterministic *or equivalently a dSHA, if it satisfies the following conditions:*

- *I contains at most one element;*
- *$\langle\rangle^\Delta$ contains at most one element;*
- *there is no epsilon transition, i.e., $\varepsilon^\Delta = \emptyset$;*
- *$a^\Delta, q^\Delta, \_^\Delta$ are partial functions from Q_h to Q_h for all $a \in \Sigma$ and $q \in Q_t$; and*
- *T^Δ is a partial function from Q_h to Q_t.*

Proposition 2. *A SHA of size n can be made deterministic in time $O(2^n)$ while preserving the hedge language.*

Proof. The determinization procedure for *SHA*s combines the determinization algorithms of word and tree automata in the natural manner, while eliminating epsilon transitions. Let ε^{Δ^*} be the reflexive and transitive closure of ε^Δ, and for any subset $Q \subseteq Q_h \cup Q_t$ let $\varepsilon^{\Delta^*}(Q) = \bigcup\limits_{q \in Q} \varepsilon^{\Delta^*}(q)$. Given a *SHA* $A = (Q_h, Q_t, \Sigma, \Delta, I, F)$, we define an equivalent deterministic *SHA* $det(A) = (Q_h^{det}, Q_t^{det}, \Sigma, \Delta^{det}, I^{det}, F^{det})$ such that $Q_h^{det} = 2^{Q_h}$, $Q_t^{det} = 2^{Q_t}$, $I^{det} = \{\varepsilon^{\Delta^*}(I)\}$ and $F^{det} = \{Q' \subseteq Q_h \mid Q' \cap F \neq \emptyset\}$. There is a unique tree initial state in

$\langle\rangle^{\Delta^{det}} = \{\varepsilon^{\Delta^*}(\langle\rangle^{\Delta})\}$ and no ε-rule, that is, $\varepsilon^{\Delta^{det}} = \emptyset$. The inference rules in Figure 14 define the missing part of Δ^{det}.

$$\frac{Q_1 \subseteq Q_h \quad P \subseteq Q_t \quad Q_2 = \{q_2 \mid \exists q_1 \in Q_1, p \in P.\ q_1 \xrightarrow{p} q_2 \in \Delta\}}{Q_1 \xrightarrow{P} \varepsilon^{\Delta^*}(Q_2) \in \Delta^{det}}$$

$$\frac{\begin{array}{c} Q_1 \subseteq Q_h \quad a \in lab(Q_1) \quad Q_2 = \{q_2 \mid \exists q_1 \in Q_1,\ q_1 \xrightarrow{a} q_2 \in \Delta\} \\ Q_2' = \{q_2 \mid \exists q_1 \in Q_1.\ q_1 \Rightarrow q_2 \in \Delta \text{ and } \nexists q_3 \in Q.q_1 \xrightarrow{a} q_3 \in \Delta\} \end{array}}{Q_1 \xrightarrow{a} \varepsilon^{\Delta^*}(Q_2 \cup Q_2') \in \Delta^{det}}$$

$$\frac{Q_1 \subseteq Q_h \quad Q_2 = \{q_2 \mid \exists q_1 \in Q_1,\ q_1 \longrightarrow q_2 \in \Delta\}}{Q_1 \longrightarrow \varepsilon^{\Delta^*}(Q_2) \in \Delta^{det}} \qquad \frac{Q_2 = \{q_2 \mid \exists q_1 \in Q_1,\ q_1 \Rightarrow q_2 \in \Delta\}}{Q_1 \Rightarrow \varepsilon^{\Delta^*}(Q_2) \in \Delta^{det}}$$

Figure 14. Determinization of *SHA*s.

We can show for all $Q_1, Q_2 \subseteq Q_h$ and $P \subseteq Q_t$ that

$$L_{Q_1,Q_2}(\det(S)) = \bigcup_{q_1 \in Q_1, q_2 \in Q_2} L_{q_1,q_2}(S)$$

so that $L_P(\det(S)) = \bigcup_{q' \in Q'} L_{q'}(S)$. Therefore, $L(det(S)) = \bigcup_{Q' \in F^{det}} L_{I,Q'}(det(S))$ and thus $L(det(S)) = \bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(S) = L(S)$. □

For illustration, the deterministic *SHA* in Figure 15 is obtained by determinization of the *SHA* in Figure 13.

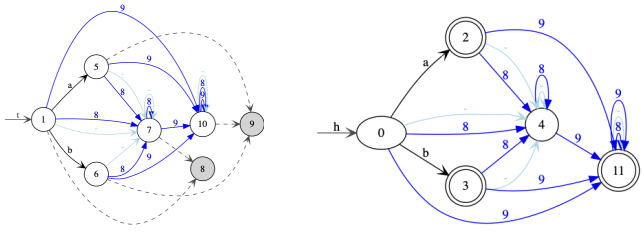

Figure 15. The determinized *SHA* $det(\text{step}(ch^*(a+b)))$.

5.2. Compilation of NREs to SHAs

As for *NWA*s, we introduce the notion of multi-module *SHA*s for which the sets of hedge states are partitioned between those that can evaluate top level positions and those to which nested positions are assigned. Therefore, multi-module *SHA*s will have exactly two modules too.

Definition 6. *A SHA* $A = (Q_h, Q_t, \Sigma, \Delta, I, F)$ *is a* multi-module *SHA if there is a subset of states* $Q_h^0 \subseteq Q_h$*, that we call* top level states*, such that*

- $I \subseteq Q_h^0$ *and*
- *the states in* Q_h^0 *can reach only other states in* Q_h^0 *via* Δ.

For instance, consider the multi-module *SHA* in Figure 13. The states of module for the top level are $Q_0 = \{1, 2, 3, 4, 5, 6, 12\}$. The others belong to the module for the nested level.

Any *NRE* E can be compiled to a multi-module *SHA* $\text{step}(E) = (Q_h, Q_t, \Sigma, \Delta, I, F)$ such that $Q_t = \{E' \mid E' = \langle E'' \rangle \text{ subexpression of } E\}$ and $L_t(E') = L(E')$ for all tree states

$E' \in Q_t$. The *SHA* step(E) can be partitioned into disjoint *SHA*s step$(E) = A^{top} \cup \bigcup_{E' \in Q_t} A^{E'}$ such that $A^{top} = (Q_h^{top}, Q_t, \Sigma, \Delta^{top}, I, F)$ and $A^{E'} = (Q_h^{E'}, Q_t, \Sigma, \Delta^{E'}, \emptyset, \emptyset)$ for all $E' \in Q_t$ and $\langle\rangle^{\Delta^{top}} = \emptyset$. Note that the transitions relation Δ is decomposed thereby into independent connected components. The automaton A^{top} can be identified with an NFA with signature $\Sigma \cup Q_t$ given that it has no tree initial states. The automata $A^{E'}$ are stepwise tree automata that recognize the tree language $L(E')$ when taking E' as final state. For this, they may have tree initial states, but will not have any initial nor final states.

Theorem 2. *For any NRE E, we can construct a SHA A such that $L(A) = L(E)$. If E contains neither conjunctions nor negations, then the construction is in time $O(|E|)$.*

Proof sketch. For the case of expressions with conjunctions or negations, the construction is analogous to the way it is done for *NWA*s. We next sketch the construction of *SHA*s for expressions without conjunction and negation.

Case $E = E' \cdot E''$**:** We use McNaughton–Yamada–Thompson algorithm [36] for composing the multi-module NFAs of step(E') and step(E''). The stepwise tree automata $A^{\langle E''' \rangle}$ of the subexpressions of type tree are preserved. For succinctness, if some subexpression $\langle E''' \rangle$ occurs more than once, then only a single copy of $A^{\langle E''' \rangle}$ is kept. References to states of the removed copy should be renamed to their equivalent counterparts.

Case $E = \langle E' \rangle$**:** We construct step(E) from step(E'). The initial states of step(E') are turned into tree initial states. We then add a new tree state $\langle E' \rangle$ and connect it to all final states of step(E') by a tree final rule $q \longrightarrow \langle E' \rangle$. Furthermore, the previously final state q becomes non final. Finally we add a new initial state q_i, a new final state q_f and a transition rule $\xrightarrow{h} (q_i) \xrightarrow{\langle E' \rangle} (q_f)$.

Case $E = \mu a.E'$**:** The main idea of the construction is similar to the case of *NWA*s. The correctness argument relies on the invariant that only multi-module *SHA*s are built.

Again by the golden lemma of the μ-calculus, we can assume w.l.o.g. that a occurs at most once in E'. By using ε-rules, we can preserve the invariant that there will be at most one pair (q, q') such that $q \xrightarrow{a} q'$ in step(E'). Furthermore, these transitions cannot be on top level, given that the occurrence of a in E' must be nested below parentheses. The automaton step(E) is obtained from step(E') by first copying the top level NFA of step(E'), as in Figure 16. We thus obtain two versions for each state of the top level NFA of step(E'): one referred to as the top level copy—$q_{0,0}$ and $q_{3,0}$ in Figure 16—, and another one as the nested level – $q_{0,1}$ and $q_{3,1}$ in Figure 16. Only top level states may be initial or final. Then, we add ε-rules from q to the nested states that correspond to the initial states of step(E'), and from the nested states corresponding to the final states of step(E') to q'. Finally we remove the rule $q \xrightarrow{a} q'$. The resulting automaton is shown in Figure 17.

Note that every transition added for a state–top level or nested—in a subsequent step of the construction—except the ε-rules added for μ-expressions—must also be added for its copy.

The construction is correct as the μ-bound name a is nested below parenthesis in E'. Therefore, it can be shown that the ε-edges introduced cannot be used to produce unwanted order in successful runs. Maintaining this invariant in polynomial time requires an additional argument. Instead of copying the top level parts of subexpressions, each state is introduced twice during the construction: one version for nesting, and another one for being part of top level parts. This way the size of the automaton is not doubled at each step, but only once.

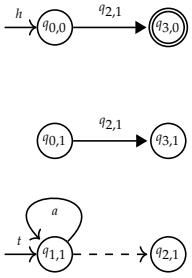

Figure 16. *SHA* for $\langle a^* \rangle$.

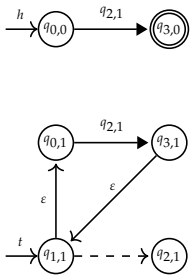

Figure 17. *SHA* for $\mu a.\langle a^* \rangle$.

We omit the correctness proof of this construction. □

Unlike *NWAs*, one cannot preserve the determinism of the expressions of $nregexp(ch, T)$ in *SHA*s, even with Glushkov-like constructions. For instance, for the deterministic *NRE* $\langle a_1 \cdot \langle a_2 \cdot \ldots \cdot \langle a_n \rangle \ldots \rangle \rangle$, one would have an *SHA* having a tree initial state for each of the $\langle a_i \ldots \rangle$ subtree, implying nondeterminism.

5.3. Experimental Results Starting with the SHA Compiler

In the first two columns of Figure 18, we report the size of the *SHA*s obtained from *NREs* by our compiler, and the size of the deterministic *SHA*s produced thereof.

	step(.)	det(step(.))	nwa(det(step(.)))	nwa(step(.))	det(nwa(step(.)))
A1	154 (56)	145 (36)	417 (37)	210 (57)	398 (37)
A2	120 (41)	427 (56)	899 (57)	177 (42)	4105 (148)
A3	128 (45)	305 (43)	622 (44)	181 (46)	907 (62)
A4	187 (66)	167 (41)	510 (42)	256 (67)	487 (42)
A5	211 (70)	411 (54)	897 (55)	298 (71)	1192 (73)
A6	284 (90)	189 (44)	587 (45)	394 (91)	548 (45)
A7	188 (64)	170 (40)	502 (41)	260 (65)	468 (41)
A8	1106 (267)	749 (123)	2831 (124)	1549 (268)	2520 (124)
B3	156 (58)	157 (35)	419 (36)	214 (59)	423 (38)

Figure 18. The *SHA*s for the benchmark *NREs* and the automata derived thereof.

The *SHA* compiler yields automata of acceptable size from the *NREs* of all benchmark queries. These sizes are given in the first column $step(.)$ of Figure 18. This even holds for A8, in contrast to the case where the produced *SHA* has overall size 1106 and 267 states.

The determinization of the *SHA*s in the second column $det(step(.))$ even yields smaller automata in all cases. For A8, we obtain a deterministic stepwise automaton of overall size 749 and with 123 states. This might be surprising, in that the determinization algorithm may lead to an exponential blow-up in the worst case. However, it may also clean the automaton, keeping only accessible sets of states. This is what seems to happen systematically on the benchmark with the exception of A2, where the size goes up by a factor of four and A5

where the size doubles. For A2 the number of states grows by one third, while for A5 it decreases by one third.

Based on the back-and-forth compiler from *SHA*s to *NWA*s from following Section 6, we can obtain deterministic *NWA*s of acceptable size for all benchmark queries. The method $nwa(det(step(.)))$ yield for A a *dNWA* of size 2831 and with 124 states. The alternative method $det(nwa(step(.)))$ yields a *dNWA* of size 2520, which is even smaller, and the same number of states.

6. NWAs versus SHAs

We next show how to compile *SHA*s to *NWA*s such that determinism is preserved, and back while introducing nondeterminism. Thereby, we can obtain small *NWA*s for *NRE*s such as $E = ch^*(a + b)$ for which $det(nwa(E))$ blew up in size in a surprising manner (see Figure 12).

6.1. SHAs to NWAs

As a first step, we introduce a transformation on *SHA*s, so that for any *SHA* A:

- if A is deterministic, the transformation returns A, and
- if A is nondeterministic with set of hedge states Q_h and transition relation Δ, the transformation returns a new *SHA* A' with set of hedge states $Q'_h = Q_h \uplus \{q_{t\text{-}init}\}$ where $q_{t\text{-}init}$ is a new hedge state, and set of transitions Δ' which equals Δ except that $\langle\rangle^{\Delta'} = \{q_{t\text{-}init}\}$ and $\varepsilon^{\Delta'} = \varepsilon^{\Delta} \cup \{(q_{t\text{-}init}, q) \mid q \in \langle\rangle^{\Delta}\}$.

Then, we compile any *SHA* $A = (Q_h, Q_t, \Sigma, \Delta, I, F)$ obtained after the above transformation to an *NWA* $nwa(A) = (Q_h, Q_t, \Sigma, \Gamma, \Delta', I, F)$ such that $L(A) = L(nwa(A))$. We set $\Gamma = Q_h$, $\_^{\Delta'} = \_^{\Delta}$, $a^{\Delta'} = a^{\Delta}$ for all $a \in \Sigma$, $\varepsilon^{\Delta'} = \varepsilon^{\Delta}$, $\mathrm{T}^{\Delta'} = \mathrm{T}^{\Delta}$:

$$\frac{q_1 \xrightarrow{\downarrow q_2} \in \Delta \quad p \in \langle\rangle^{\Delta}}{q_1 \xrightarrow{\downarrow q_1} p \in \Delta' \text{ and } q \xrightarrow{\uparrow q_1} q_2 \in \Delta'}$$

Clearly, if S is deterministic then so is $nwa(S)$, as p is unique in this case in particular. One might be tempted to skip the first-step transformation and restrict the above construction rule to states p such that $L_q(S[\langle\rangle^{\Delta}/\{p\}]) \neq \emptyset$. However, this would lead to a huge blow-up when determinizing these *NWA*s, basically as this change spoils the single-entry property discussed in Definition 7.

The conversion of step$(ch^*(a + b))$ in Figure 13 yields the *NWA* in Figure 19. Note that the opening rules are deterministic (but not the whole *NWA*), as for all tree states q there is at most one hedge state p with $\langle\rangle \to p$ such that q is accessible from p. The NWA has size 64, while its determinization has size 159 (see Figure A4 of the Appendix C). The size increase raised by determinization is thus $95 = 159 - 64$ for this *NWA*.

The conversion of step$(ch^*(a + b))$ in Figure 13 yields the *NWA* in Figure 19. Note that the opening rules are deterministic (but not the whole *NWA*), as for all tree states q there is at most one hedge state p with $\langle\rangle \to p$ such that q is accessible from p. The NWA has size 64, while its determinization has size 159 (see Figure A4 of the Appendix C). The size increase raised by determinization is thus $95 = 159 - 64$ for this *NWA*.

The size increase for determinization is considerably smaller for the *NWA* obtained from the regular expressions by indirection via a *SHA* than for *NWA*s obtained by direct compilation. Indeed, the determinization of $nwa(ch^*(a + b))$ blows the size from 39 to 271. The size increase for the determinization of $nwa(ch^*(a + b))$ is thus $242 = 271 - 39$ while for $nwa(\text{step}(ch^*(a + b)))$ it is only $95 = 159 - 64$.

The experiments will show that this is not an exception but the general rule. Intuitively, the reason is that *NWA*s obtained from *SHA*s do all the work bottom-up, where *NWA*s obtained directly from the regular expression do a considerable amount of work top-down. In terms of the work in [22], this restriction can be characterized syntactically by the single-entry property:

Definition 7. *A (weak) single-entry NWA* $A = (Q_h, Q_t, \Sigma, \Gamma, \Delta, I, F)$ *is a NWA for which there exists a single state* $q_{entry} \in Q_h$ *such that all opening rules in* Δ *have the form* $q \xrightarrow{\downarrow q} q_{entry}$.

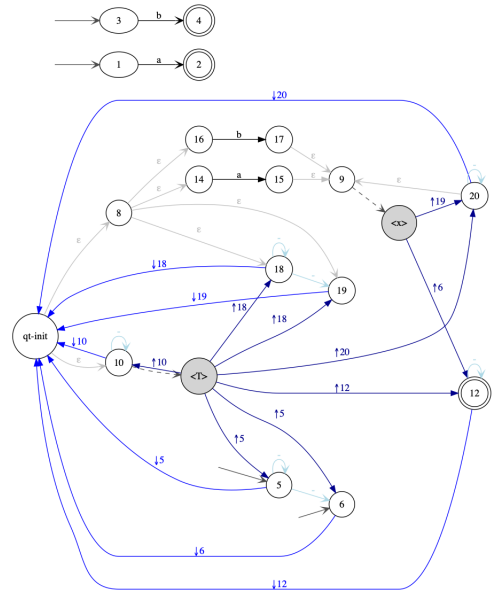

Figure 19. The single-entry *NWA* $nwa(\text{step}(ch^*(a+b)))$ obtained from the *SHA*.

Note that *call-driven automata* (CDAs) discussed in [23] coincide with multi-module single-entry *dNWAs* and also with *(multi-module) single-entry visibly pushdown automata* [22,24].

It can be shown that $nwa(S)$ has the (weak) single-entry property for all *SHA*s S for which the p's are unique in the above construction rule, i.e., such that $\langle\rangle^{\Delta} = \{p\}$. Note that this was not the case for $\text{step}(ch^*(a+b))$ in Figure 13 but could have been imposed w.l.o.g. leading to a slightly different *NWA* than in Figure 19.

The conversion of the determinization $det(\text{step}(ch^*(a+b)))$ in Figure 15 yields the deterministic *NWA* in Figure 20. The size goes up slightly from 53 to 73. It should be noticed that factorization avoids a quadratic blow up in this case. This can be observed at state 14, which has 3 incoming tree edges and 10 outgoing closing edges. Without factorization, the 3 tree edges could be replaced by 3 ε-edges whose elimination would produce 30 closing edges. This would increase the number $3 + 10$ edges to $3 * 10$ edges.

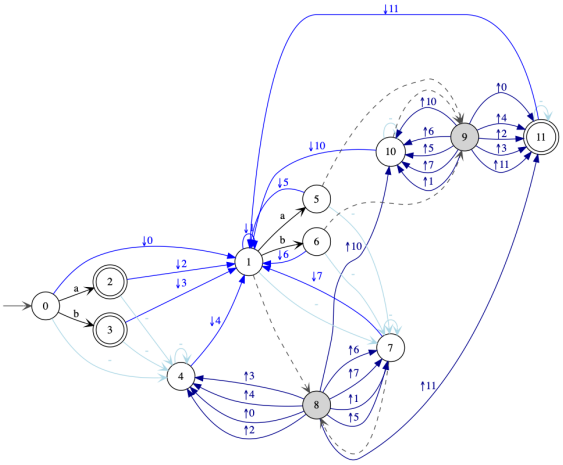

Figure 20. Deterministic *NWA*: $nwa(det(\text{step}(ch^*(a+b))))$.

6.2. NWAs to SHAs

Conversely, *NWAs* can be compiled to stepwise hedge automata, but at the cost of introducing nondeterminism, as an *NWA* may traverse the branches of a tree top-down, while a stepwise must traverse them bottom-up. For this, the stepwise guesses the state in which the *NWA* will arrive from above and then evaluates the subtree starting with this state, while verifying the correctness of the guess later on. Let $A = (Q_h, Q_t, \Sigma, \Delta', I, F)$ be an *NWA*. We build a *SHA* $\text{step}(A) = (Q_h^s, Q_t^s, \Sigma, \Delta^s, I^s, F^s)$ where $Q_h^s = Q_h \times Q_h$, $Q_t^s = Q_h \times Q_t$, $I^s = \{(q,q) \mid q \in I\}$, $F^s = I \times F$ and Δ^s is the smallest satisfying the following rules:

$$\frac{o \in \Sigma \cup \{\_, \varepsilon\} \quad q_1 \xrightarrow{o} q_2 \in \Delta \quad q \in Q_h}{(q,q_1) \xrightarrow{o} (q,q_2) \in \Delta^s} \qquad \frac{q_1 \dashrightarrow q_2 \in \Delta \quad q \in Q_h}{(q,q_1) \dashrightarrow (q,q_2) \in \Delta^s}$$

$$\frac{q_1 \xrightarrow{\downarrow\gamma} q_2 \in \Delta}{(q_2,q_2) \in \langle\rangle^{\Delta^s}} \qquad \frac{q_1 \xrightarrow{\downarrow\gamma} q_2 \in \Delta \quad q_3 \in Q_t \quad q_3 \xrightarrow{\uparrow\gamma} q_4 \in \Delta \quad q \in Q_h}{(q,q_1) \xrightarrow{(q_2,q_3)} (q,q_4) \in \Delta^s}$$

The construction is such that $L(A) = L(\text{step}(A))$.

For the *NWA* $nwa(ch^*(a+b))$ in Figure 6, we obtain the stepwise in Figure A3 up-to removing useless states and separating the top level. Determinization yields $det(\text{step}(nwa(ch^*(a+b)) = det(\text{step}(ch^*(a+b)))$ in Figure 15.

7. Optimizations

We will use three optimization methods for constructing smaller *dSHAs* and thus smaller *dNWAs*: minimization, symbolic representations of sets of transition rules, and schema-based cleaning.

7.1. Minimization

Our next objective is to reduce the size of deterministic *SHAs* by developing a minimization algorithm for a subclass of *dSHAs*. Even though our implementation can deal with them, we consider *SHAs* without symbolic rules $q \xrightarrow{} q'$ for simplicity in this section.

We start with an example that motivates the choice of our subclass. In Figures 21 and 22, two *dSHAs* are given that both recognize the language of all hedges with signature $\Sigma = \{x, y\}$ containing exactly one occurrence of the letter x. The *dSHA* in Figure 22 is the *dSHA* recognizing this language which has the minimal number of states. The *dSHA* in Figure 21 is the minimal multi-module *dSHA* for this language. The question is how a minimization algorithm for *dSHAs* could convert the *dSHA* in Figure 21 to this minimal one in Figure 22. In particular, why would it merge the tree initial state and the hedge initial state? We do not see how this could be done based on some Myhill–Nerode-like equivalence relation. This motivates an a priori restriction to *dSHAs* imposing that the tree initial state and hedge initial state must be equal.

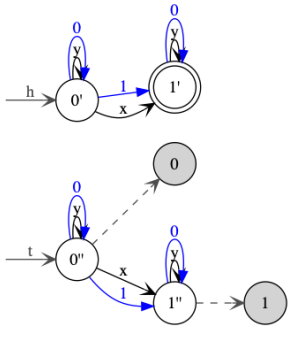

Figure 21. A *dSHA* for hedges over $\Sigma = \{x, y\}$ with single occurrence of x. It is minimal in the class of multi-module *dSHAs*.

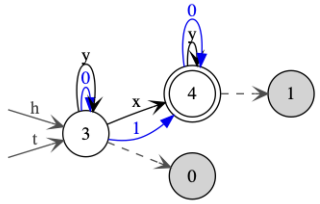

Figure 22. An equivalent *dSHA* to that in Figure 21 that is minimal in the class of *dSHA*s with equal tree and hedge initial states.

Note that any *SHA* can be converted into a *dSHA* with equal tree and hedge initial states. For this, it is sufficient to "fuse" these states and then to determinize the *SHA* obtained. When doing so for the *dSHA* in Figure 21, we indeed obtain the minimal *dSHA* from Figure 23, so no further minimization is needed in this case.

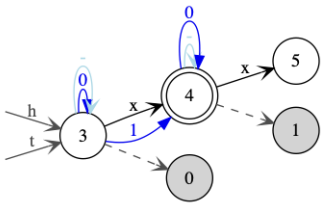

Figure 23. A single x-marked position.

Given the close relationship between *SHA*s and weak single-entry *NWA*s, it is instructive to consider the existing results on minimization for *dNWA*s. It is known that the class of general *dNWA*s does not allow for unique minimization [22] and that the minimization becomes NP-hard when admitting general signatures with multiple parenthesis [24].

On the positive side, the best existing minimization algorithm is due to Chervet and Walukiewicz [23]. It applies to the subclass of *multi-module single-entry dNWA*s, called there *call-driven automata (CDA)* (Chervet and Walukiewicz [23] permit signatures with multiple opening parenthesis. In the case of a single opening parenthesis, the class of CDAs is equal to their subclass of *expanded CDA*s for which they develop their minimization algorithm in the first place.). They showed that the subclass of multi-module single-entry *dNWA*s enjoys unique minimization in polynomial time.

In the case of *dSHA*s, we believe that unique minimization holds for the following two subclasses, and will show it for the second:

- the subclass of multi-module *dSHA*s, and
- the subclass of *dSHA*s where the hedge and tree initial state are the same, i.e., $\langle\rangle^{\Delta} = I$.

The first subclass of multi-module *dSHA*s is motivated by the subclass of multi-module single-entry *dNWA*s. Note, however, that the *SHA*s that are obtained by compilation from single-entry *dNWA*s need not be deterministic, so the analogy between both automata classes is not perfect. The *dSHA* in Figure 21 is minimal for the class of multi-module *dSHA*s.

The second subclass of *dSHA*s corresponds to the subclass of single-entry *dNWA*s in which the single-entry state is equal to the initial state. The *dSHA* in Figure 22 is minimal for the second subclass. In the remainder of this section, we present a minimization algorithm for the second subclass. For this, we identify *dSHA*s in which tree and hedge initial state coincide with two-sorted deterministic tree automata, so that we can use a minimization algorithm for the latter. Our automaton translation is based on a novel encoding of hedges into ranked well-sorted trees with monadic and binary function symbols, which is inspired by the previous binary encoding of unranked trees known from stepwise tree automata [21]. For any unranked signature Σ, as for the construction of hedges, we consider two sorts:

$\mathtt{h}$ for hedges and $\mathtt{t}$ for trees. We then consider the following ranked signature with these two sorts:

$$\Sigma_@ = \{a^{(\mathtt{h})} \mid a \in \Sigma\} \cup \{@^{(\mathtt{h}\times\mathtt{t}\to\mathtt{h})}, \varepsilon^{(\mathtt{h})}, T^{(\mathtt{h}\to\mathtt{t})}\}$$

The well-sorted trees over $\Sigma_@$ of both sorts then have the following abstract syntax:

$$\begin{array}{lll} \text{well-sorted trees of sort } \mathtt{h}: & \tau & ::= \quad a(\tau) \mid @(\tau,\tau') \mid \varepsilon \\ \text{well-sorted trees of sort } \mathtt{t}: & \tau' & ::= \quad T(\tau) \end{array}$$

Any hedge over Σ can be encoded into a ranked well-sorted tree of sort $\mathtt{h}$ with signature $\Sigma_@$. For instance, the hedge

$$h = \langle a \cdot \langle b \cdot \langle c \cdot d \cdot e \rangle \rangle \cdot f \rangle$$

is encoded into the following ranked well-sorted tree of sort $\mathtt{h}$ over $\Sigma_@$:

$$[[h]] = \varepsilon @ T(f(a(\varepsilon)@\tau')) \quad \text{where} \quad \tau' = T(b(\varepsilon)@\tau'') \quad \text{and} \quad \tau'' = T(e(d(c(\varepsilon))))$$

Any *SHA* $A = (Q_h, Q_t, \Sigma, \Delta, I, F)$ with equal tree and hedge initial states, that is, $\langle\rangle^\Delta = I$, can then be encoded into a two sorted tree automaton $[[A]] = (Q_h, Q_t, \Sigma_@, \Delta', I, F)$ by mapping the transition rules in Δ to those in Δ' as follows:

Δ	Δ'
$q \xrightarrow{a} q'$	$a(q) \to q'$
$q \longrightarrow p$	$T(q) \to p$
$q \xrightarrow{p} q'$	$q@p \to q'$
$q_i \in \langle\rangle^\Delta = I$	$\varepsilon \to q_i$

We can first note that $[[L(A)]] = L([[A]])$. Second, the translation function is a bijection between *SHA*s over Σ and two-sorted tree automata over $\Sigma_@$. Furthermore, this translation preserves determinism. It follows that if A is a *dSHA* with a minimal number of states recognizing $L(A)$ then $[[A]]$ is a deterministic two-sorted tree automaton with a minimal number of states recognizing $[[L(A)]]$. Furthermore, the unique minimization of deterministic two-sorted tree automata implies the unique minimization of the class of *dSHA*s with equal tree and hedge initial states.

Using this translation back and forth, we can thus lift the minimization algorithm of deterministic two-sorted tree automata to a minimization algorithm for the subclass of *dSHA*s with equal tree and hedge initial states. This is the minimization algorithm for *dSHA*s that we have implemented. We then used it in our constructions to reduce the size of the *dSHA*s obtained by determinization.

7.2. Symbolic SHAs with Apply-Else Rules

The sizes of the *dSHA*s constructed so far are dominated by the number of transitions. We now propose a class of symbolic *dSHA*s by adding apply-else rules, in order to represent large numbers of apply rules in a more compact and symbolic manner.

An apply-else rule has the form $q \xrightarrow{-} q'$ where $q, q' \in Q_h$. It represents the set of apply rules $q \xrightarrow{p} q'$, where $p \in Q_t$ can be chosen arbitrarily from a subset of tree states distinguished by the automaton.

We have also adapted our determinization for *dSHA*s so that it preserves apply-else rules. What is missing so far is a concept for *NWAs* that corresponds to the apply-else rules of *dSHA*s. Therefore, we have to eliminate apply-else rules before translating *SHA*s to *NWAs*.

7.3. Schema-Based Cleaning

Automata for XPATH queries recognize nested words that can be obtained by encoding XML documents with a single x-marked position. The class of such nested words is characterized by a schema that we can define as the intersection of the two *dSHA*s in Figures 23 and 24. The first *SHA* tests whether there is exactly one occurrence of the internal letter x, and the second one tests that the XML data model is satisfied, and the node annotations with x and $\neg x$ are put at the right positions.

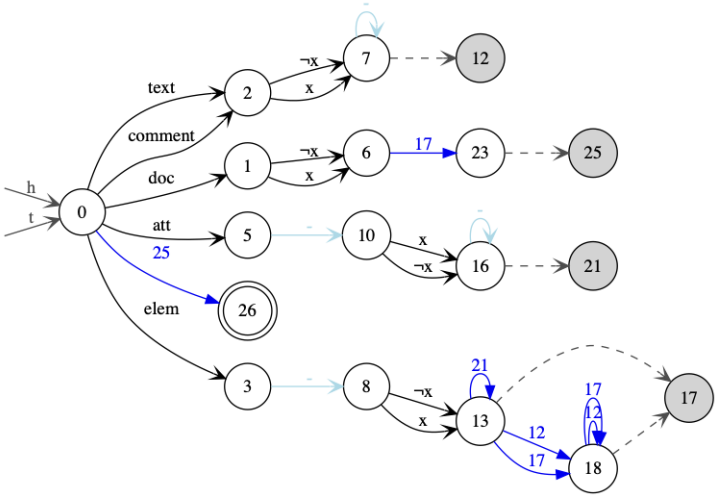

Figure 24. Nested words of x-marked XML documents.

The automata constructed for the XPATH queries may accept some trees that do not satisfy the schema (but will never be evaluated on such trees when answering the query). The idea of schema-based cleaning is to remove all transition rules and states that are not used for recognizing any nested word satisfying the schema. Schema-based cleaning of an automaton can be performed by constructing the product of the automaton with the schema, which is in our case an intersection of two *dSHA*s. We then only keep those states of the original *SHA* that are used in accessible and co-accessible states of the product with the schema.

Note that schema-based cleaning typically changes the language of the automaton. Different languages may be obtained when cleaning different automata for the same query with respect to the schema. If one is interested in a unique language, then one can choose the intersection of the automaton with the schema. This intersection, however, is usually larger than the automaton obtained by schema-based cleaning.

7.4. Experimental Results for Optimizations

The sizes of optimized automata for the benchmark queries are reported in Figure 25.

	step$_{@_}$	D(.)=det(step$_{@_}$(.))	S(.)=schema-clean(D(.))	M(.)=mini(S(.))	elim$_{@_}$(M(.))	nwa(M(.))
A1	133 (56)	145 (36)	106 (36)	106 (36)	234 (36)	268 (37)
A2	104 (41)	157 (30)	101 (30)	55 (16)	73 (16)	87 (17)
A3	111 (45)	193 (32)	123 (32)	95 (24)	145 (24)	167 (25)
A4	160 (66)	167 (41)	123 (41)	123 (41)	294 (41)	334 (42)
A5	179 (70)	387 (53)	274 (53)	274 (53)	580 (53)	650 (54)
A6	237 (90)	189 (44)	144 (44)	144 (44)	364 (44)	410 (45)
A7	159 (64)	166 (40)	125 (40)	115 (36)	241 (36)	279 (37)
A8	894 (267)	639 (117)	527 (117)	487 (101)	1257 (101)	1413 (102)
B3	139 (58)	135 (33)	102 (33)	96 (32)	200 (32)	228 (33)

Figure 25. Optimized automata for derived stepwise automata compiled from *NREs*.

The function $step_{@\_}$ used in the first column compiles *NREs* to *SHA*s with apply-else rules. This does not change the number of states, but reduces the number of automata transitions. In the case of A8, the size of the stepwise automaton is reduced from 1106 to 894.

An optimized determinizer is applied by the function $D(.) = det(step_{@\_}(.))$ in the second column. It preserves apply-else rules in particular. For A8, the size is reduced from 749 to 639 while the number of states is preserved.

Schema-based cleaning is applied by the function $S(.) = schema - clean(D(.))$ in the third column. For A8, the number of rules is reduced further from 639 to 527.

Minimization is applied by the function $M(.) = mini(S(.))$ in the fourth column. In the case of A8, it reduces the number of states from 117 to 101 and the size from 527 to 487.

In order to come back to *dNWAs*, we have to eliminate the apply-else rules in column six. For A8 this increases the number of rules back from 527 to 1257.

In the final column, we apply the compiler from *SHA*s to *NWAs* which preserves determinism. For *A*8, this results in a *dNWA* of size 1413 and 102 states. This is better than the previous results, in particular with respect to the number of states.

8. Summary of Experimental Results

We now plug the different compilers and optimization methods all together and compare the sizes of deterministic *NWAs* that we can obtain thereby.

The overall sizes (#states) of the resulting *dNWAs* are given in Figure 26. We see that the two methods starting with *SHA*s $nwa(det(step(.)))$ and $det(nwa(step(.)))$ yield reasonably small deterministic *NWAs* for the *NREs* of all benchmark queries. The methods starting with *NWAs* $nwa(det(step(nwa(.))))$ and $det(nwa(step(nwa(.))))$ provide reasonably small deterministic *NWAs* for queries except for A8.

	det(*nwa*(.))	*nwa*(*det*(step(.)))	*det*(*nwa*(step(.)))	*nwa*(*det*(step(*nwa*(.))))	*det*(*nwa*(step(*nwa*(.))))
A1	—	268 (37)	363 (37)	204 (37)	363 (37)
A2	362,600 (6782)	87 (17)	3781 (142)	67 (17)	540 (51)
A3	318,704 (8216)	167 (25)	837 (61)	113 (25)	417 (43)
A4	—	334 (42)	447 (42)	298 (42)	447 (42)
A5	—	650 (54)	1110 (72)	194 (34)	612 (54)
A6	—	410 (45)	507 (45)	349 (45)	507 (45)
A7	—	279 (37)	431 (41)	162 (30)	431(41)
A8	—	1413 (102)	2406 (124)	—	—
B3	—	228 (33)	392 (38)	189 (33)	392 (38)

Figure 26. Deterministic *NWAs* computed with optimizations for the XPath benchmark queries. Note that different *dNWAs* for the same query may recognize different languages, due to schema-based cleaning with respect to the XML data model. Furthermore, our implementation of the minimization algorithm for the subclass of *dSHA*s worked successfully only for *dSHA*s with at most 200 states.

We also tested our algorithms on collections of XPath queries with a scalable parameter, such as the queries $ch^n(a)$ for increasing n. This series is known to require automaton with a number of states exponential in n for deterministic bottom-up evaluation. The best methods to produce deterministic *NWAs* in this case is $nwa(det(\text{step}))$. It works until $n = 9$, leading to an *dNWA* of size 134,929 with 772 states. The number of states close to doubles when increasing n by 1. The second best method for producing *dNWAs* for the series $ch^n(a)$ works only until $n = 6$.

For explaining the different size of the *dNWAs* for the series $ch^n(a)$, we first note that no schema-based cleaning was applied in this experiment. As a consequence, unique

minimal single-entry *dNWAs* in which the single-entry state is the initial state should exist. The reason for the larger number of states with the three other methods is that we have not implemented any minimization algorithm for this subclass of single-entry *dNWAs*. Furthermore, our implementation of the minimization algorithm for our subclass of *dSHA*s failed for too big *dSHA*s. In this case, the number of states reported in Figure 27 could not be reduced to the minimum. In addition, the number of rules seems to be increased further by the lack of any symbolic representation for rules of *NWAs* that could mimic the apply-else rules for *SHA*s.

	det(*nwa*(.))	*nwa*(*det*(step(.)))	*det*(*nwa*(step(.)))	*nwa*(*det*(step(*nwa*(.)))	*det*(*nwa*(step(*nwa*(.))))
$ch^0(a)$	4 (2)	4 (2)	4 (2)	4 (2)	4 (2)
$ch^1(a)$	165 (33)	34 (7)	55 (10)	34 (7)	55 (10)
$ch^2(a)$	1530 (199)	55 (10)	112 (16)	55 (10)	112 (16)
$ch^3(a)$	198,28 (1281)	109 (16)	352 (32)	109 (16)	352 (32)
$ch^4(a)$		265 (28)	2200 (88)	265 (28)	2200 (88)
$ch^5(a)$		769 (52)	22,792 (296)	769 (52)	22,792 (296)
$ch^6(a)$		2545 (100)	303,592 (1096)	80,369 (2148)	
$ch^7(a)$		9169 (196)			
$ch^8(a)$		34,705 (388)			
$ch^9(a)$		134,929 (772)			

Figure 27. Deterministic *NWAs* for the queries $ch^n(a)$ where $n = 0, \ldots, 9$: size (#states). There is no schema-based cleaning. Our implementation of the minimization algorithm was applied to all *dSHA* with at most 200 states, as it failed for larger *dSHA*s. No minimization algorithm for subclasses of single-entry *dNWAs* was implemented.

9. Deterministic Nested Regular Expressions

We finally show how to distinguish *NREs* that can be evaluated deterministically in polynomial time, for instance, by compilation to deterministic *NWAs*. For this, we consider the language of *NREs* $nregexp(ch, T)$ that extends the abstract syntax of *NREs* by a new constant T and a new unary operator ch.

Definition 8. *An expression of* $nregexp(ch, T)$ *is deterministic if it does not contain a subexpression of any of the forms:* $E_1 + E_2$, E^*, $T \cdot E$, $\mu a.E$.

Note in particular that $ch(a)$ is a deterministic expression of $nregexp(ch, T)$, as the child operator is added as a primitive there. In contrast, the semantically equivalent expression $T.\langle a \rangle.T$ is not deterministic. Similarly, T is deterministic since it is a primitive expression of $nregexp(ch, T)$, while the equivalent expression $\mu x.(\langle x \rangle + \_)^*$ is nondeterministic for three different reasons: the μ-operator, star $*$, and disjunction $+$. The recursive expression $ch^*(E)$ is nondeterministic: it is not primitive in $nregexp(ch, T)$, and its definition is based on the μ-operator and disjunction.

The only query of the benchmark for which we can provide a deterministic *NRE* is the query A1. The *NRE* for query A1 in Figure A1 is nondeterministic nevertheless, as we replaced $ch(E)$ with $T \cdot \langle E \rangle \cdot T$. This is not problematic, given that we can use a decent method for determinization of *NWAs*. For this reason, it does no more seem worth the effort to maintain specialized compilation methods for deterministic *NREs*. For the same reason, we will not present any experimental results for our specialized compiler from deterministic *NREs* to deterministic *NWAs*. Instead we use the more general compiler for general nondeterministic *NREs*.

The compiler from Theorem 1 introduces epsilon rules, and thus it does not preserve determinism: some deterministic *NREs* will be compiled to nondeterministic *NWAs*.

This introduction of nondeterminism can be avoided by eliminating epsilon rules on the fly, that is by using Glushkov's approach rather than that of Thompson.

Theorem 3. *For any deterministic regular expression E of $nregexp(ch, T)$ without conjunction and negation, we can construct in time $O(|E|^2)$ a dNWA recognizing the same language.*

Proof sketch. Theorem 3 uses Glushkov's construction and thus eliminates ε-edges on the fly compared to the McNaughton–Yamada–Thompson algorithm. The Glushkov construction is well-known to preserve determinism when compiling regular expressions without nesting to finite state automata [20]. For the additional deterministic expressions $ch(E)$, we adapt the deterministic compilation from the work in [18]. This quadratic time result generalizes a previous result for the Glushkov construction [19] from regular expressions without conjunctions and negations to *NREs* without conjunctions and negations. □

Small deterministic *NREs* without conjunction and negation can thus be compiled to small *dNWAs*. On the benchmark, however, this construction can be applied to the query $A1$ only, so only a few queries can be covered in this manner.

10. Conclusions and Future Work

We presented *SHA*s and showed how they can be used to compile *NREs* to deterministic *NWAs*. When applied to *NREs* for navigational XPATH queries in the usual XPathMark benchmark, we obtained reasonably small deterministic *NWAs*, in contrast to all previous approaches.

The *dNWAs* that we obtain by compilation from *SHA*s all have the weak single-entry property. This property means that the computation of the *NWA* is done in a purely bottom-up and left-to-right manner, so in the same way as by an *SHA*. Our experiments show that the usual determinization algorithm for *NWAs* is well-behaved when applied to weak single-entry *NWAs*, while it quickly fails without the weak single-entry property.

We have also stated a unique minimization algorithm for *dSHA*s with the same tree and hedge initial state. It is open whether unique minimization holds for general *dSHA*s. Neither do we know whether *dSHA* minimization is NP-hard. The analogous questions remain open for the class of weak single-entry *dNWAs*.

In future work, one needs to tackle the open questions on the minimization of *dSHA*s, weak single-entry *dNWAs*, and *dNWAs* with fixed general signatures. One has to understand, whether and why unique minimization holds or not, and whether and why minimization is hard or not. Independently, it is interesting to use *SHA*s in various questions in theory and practice. In particular, we want to develop new algorithms for earliest query answering for *dSHA*s that are more efficient than the existing algorithms for *dNWAs* [16] and to see how they behave in practice.

Author Contributions: Conceptualization, J.N.; methodology, J.N.; software, J.N. and M.S.; validation, J.N. and M.S.; formal analysis, J.N. and M.S.; writing—original draft preparation, J.N. and M.S.; writing—review and editing, J.N. and M.S.; visualization, J.N. and M.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by a grant from CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015–2020.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The nested regular expressions generated for the queries mentioned in this article and their corresponding automata—when we could produce them—can be found at http://researchers.lille.inria.fr/niehren/complementary-material.

Acknowledgments: It is a pleasure to thank Iovka Boneva for her contributions to the conference version [25] onto which this journal article extends. We are equally grateful to Antonio Al Serhali for implementing the determinization algorithm for *SHA*s with apply-else rules.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Determinization of *NWAs*

Let us first introduce some notations. For a transition $\tau \in Q_h \times Q_h$, we write $lab(\tau) = \{a \in \Sigma \mid \exists (q, q') \in \tau, q'' \in Q. q' \xrightarrow{a} q'' \in \Delta\}$. Furthermore, we write ε^{Δ^*} to denote the reflexive and transitive closure of ε^{Δ}. Finally, for any set Q, we write id_Q to denote the binary relation that relates every element of Q to itself, that is, $id_Q = \{(q, q) \in Q^2\}$.

We adapt the usual determinization procedure for *NWAs* [3,18] so that they can account for hedge ending and else rules. Given an *NWA* $A = (Q_h, Q_t, \Sigma, \Gamma, \Delta, I, F)$, the difficulty is to deal with concurrent opening rules $q \xrightarrow{\downarrow\gamma_1} q_1$ and $q \xrightarrow{\downarrow\gamma_2} q_2$ in Δ during determinization without mixing up the stack symbols γ_1 and γ_2. Therefore, we use transition relations as states of the determinized automaton $\det(A) = (Q_h^{det}, Q_t^{det}, \Sigma, \Gamma^{det}, \Delta^{det}, I^{det}, F^{det})$, that is, $Q_h^{det} = 2^{Q_h \times Q_h}$, $Q_t^{det} = 2^{Q_h \times Q_t}$. The only initial state is the composition of id_I with ε^{Δ^*}, i.e., $I^{det} = \{id_I \circ \varepsilon^{\Delta^*}\}$. The set of final states is $F^{det} = \{\tau \in Q_h^{det} \mid \tau \cap (I \times F) \neq \emptyset\}$. Schemas generating the transition rules in Δ^{det} are given below.

$$\frac{\tau \in Q_h^{det}}{\tau \Rightarrow \tau \circ \_^{\Delta} \circ \varepsilon^{\Delta^*} \in \Delta^{det}} \qquad \frac{\tau \in Q_h^{det} \quad Q' = \{q' \mid \exists (\_, q) \in \tau.\, q \xrightarrow{\downarrow\gamma} q' \in \Delta\}}{\tau \xrightarrow{\downarrow\tau} id_{Q'} \circ \varepsilon^{\Delta^*} \in \Delta^{det}}$$

$$\frac{\tau \in Q_h^{det}}{\tau \xrightarrow{tree} \tau \circ tree^{\Delta} \circ \varepsilon^{\Delta^*} \in \Delta^{det}} \qquad \frac{\tau \in Q_t^{det} \quad \tau' \in Q_h^{det} \quad \tau'' = \bigcup_{\gamma \in \Gamma} \langle {}^{\Delta}_{\gamma} \circ \varepsilon^{\Delta^*} \circ \tau \circ \rangle^{\Delta}_{\gamma}}{\tau \xrightarrow{\uparrow\tau'} \tau' \circ \tau'' \circ \varepsilon^{\Delta^*} \in \Delta^{det}}$$

$$\frac{\tau \in Q_h^{det} \quad a \in lab(\tau) \quad \tau' = \{(q, q') \in \_^{\Delta} \mid \nexists q''.q \xrightarrow{a} q'' \text{ wrt} \Delta\}}{\tau \xrightarrow{a} \tau \circ (a^{\Delta} \cup \tau') \circ \varepsilon^{\Delta^*} \in \Delta^{det}}$$

Appendix B. NREs for the XPathMark Benchmark

We compiled navigational XPath queries of the XPathMark benchmark to the *NREs* given in Figure A1.

A1 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot closed\_auctions \cdot \_ \cdot T \cdot$
$\langle elem \cdot closed\_auction \cdot \_ \cdot T \cdot \langle elem \cdot annotation \cdot \_ \cdot T \cdot$
$\langle elem \cdot description \cdot \_ \cdot T \cdot \langle elem \cdot text \cdot \_ \cdot T \cdot$
$\langle elem \cdot keyword \cdot x \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle$

A2 $\langle doc \cdot \_ \cdot (\mu d_1.(\langle elem \cdot closed\_auction \cdot \_ \cdot T \cdot (\mu d_2.($
$\langle elem \cdot keyword \cdot x \cdot T\rangle \cdot T + \langle T \cdot d_1 \cdot T\rangle \cdot T))\rangle \cdot T + \langle T \cdot d_1 \cdot T\rangle \cdot T))\rangle$

A3 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot closed\_auctions \cdot \_ \cdot T \cdot \langle elem \cdot closed\_auction \cdot \_ \cdot T \cdot ($
$\mu d.(\langle elem \cdot keyword \cdot x \cdot T\rangle \cdot T + \langle T \cdot d \cdot T\rangle \cdot T))\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle$

A4 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot$
$\langle elem \cdot closed\_auctions \cdot \_ \cdot T \cdot \langle elem \cdot closed\_auction \cdot \_ \cdot$
$(T \cdot \langle elem \cdot annotation \cdot \_ \cdot T \cdot \langle elem \cdot description \cdot \_ \cdot T \cdot$
$\langle elem \cdot text \cdot \_ \cdot T \cdot \langle elem \cdot keyword \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T)$
$\&\ (T \cdot \langle elem \cdot date \cdot x \cdot T\rangle \cdot T\rangle \cdot T)\rangle \cdot T\rangle \cdot T\rangle$

A5 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot closed\_auctions \cdot \_ \cdot T \cdot ($
$\langle elem \cdot closed\_auction \cdot \_ \cdot T \cdot ($
$\mu d.(\langle elem \cdot keyword \cdot \_ \cdot T\rangle \cdot T + \langle T \cdot d \cdot T\rangle \cdot T))\rangle \cdot T)$
$\&(\langle elem \cdot closed\_auction \cdot \_ \cdot T \cdot \langle elem \cdot date \cdot x \cdot T\rangle \cdot T\rangle \cdot T)\rangle \cdot T\rangle \cdot T\rangle$

A6 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot people \cdot \_ \cdot T \cdot ((\langle elem \cdot person \cdot \_ \cdot T \cdot$
$\langle elem \cdot profile \cdot \_ \cdot T \cdot \langle elem \cdot gender \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T)$
$\&(\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot profile \cdot \_ \cdot T \cdot \langle elem \cdot age \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T\rangle \cdot T))$
$\&(\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot name \cdot x \cdot T\rangle \cdot T\rangle \cdot T)\rangle \cdot T\rangle \cdot T\rangle$

A7 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot people \cdot \_ \cdot T \cdot ((\langle elem \cdot person \cdot \_ \cdot T \cdot$
$\langle elem \cdot phone \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T+$
$\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot homepage \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T))$
$\&(\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot name \cdot x \cdot T\rangle \cdot T\rangle \cdot T)\rangle \cdot T\rangle \cdot T\rangle$

A8 $\langle doc \cdot \_ \cdot \langle elem \cdot site \cdot \_ \cdot T \cdot \langle elem \cdot people \cdot \_ \cdot T \cdot$
$((\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot address \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T)$
$\&\ (\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot phone \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T +$
$\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot homepage \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T))$
$\&\ ((\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot creditcard \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T +$
$\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot profile \cdot \_ \cdot T\rangle \cdot T\rangle \cdot T)))$
$\&\ (\langle elem \cdot person \cdot \_ \cdot T \cdot \langle elem \cdot name \cdot x \cdot T\rangle \cdot T\rangle \cdot T)\rangle \cdot T\rangle \cdot T\rangle$

Figure A1. The *NREs* of the XPath benchmark queries.

Appendix C. Some More Automata

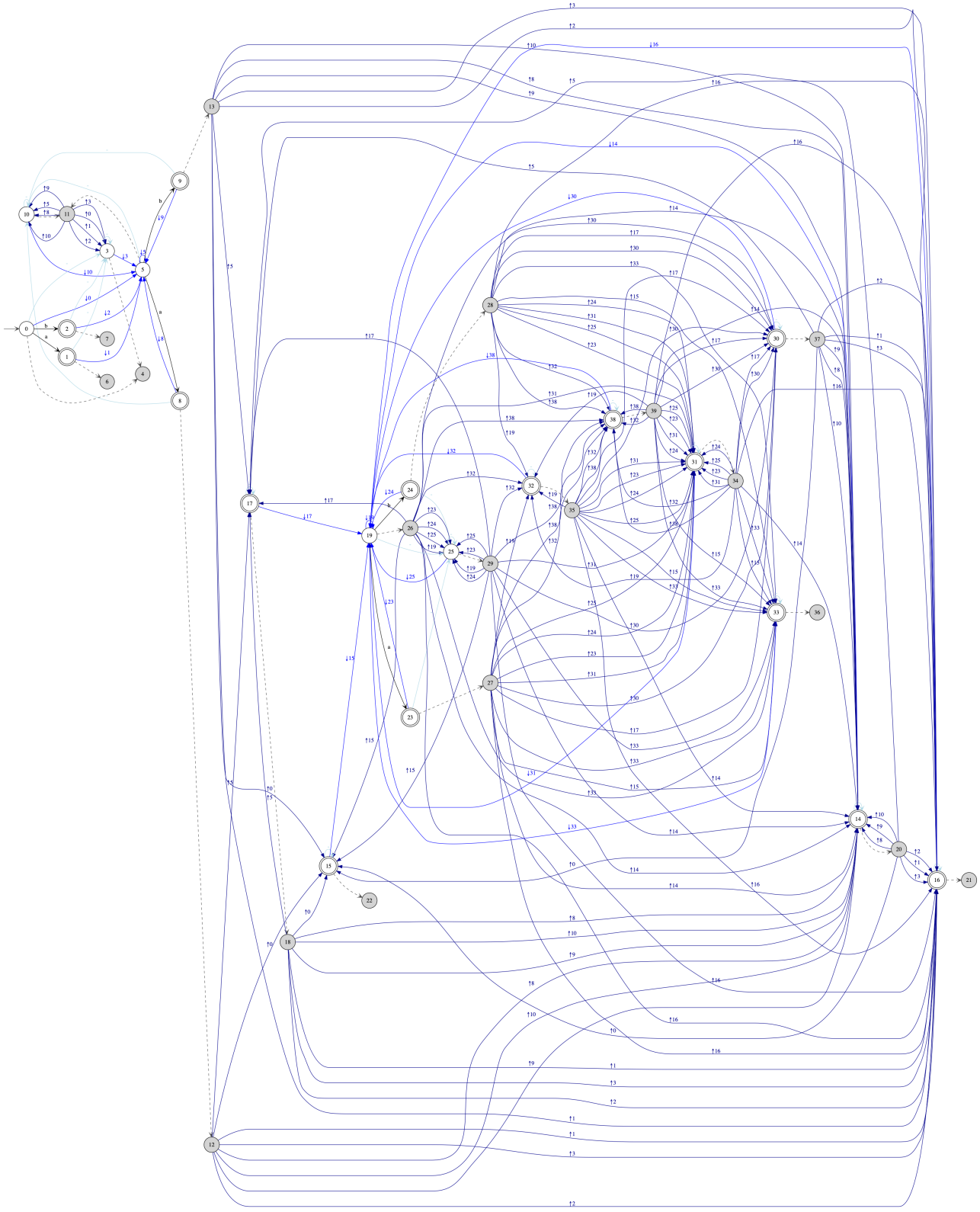

Figure A2. Deterministic *NWA*: $det(nwa(ch^*(a+b)))$.

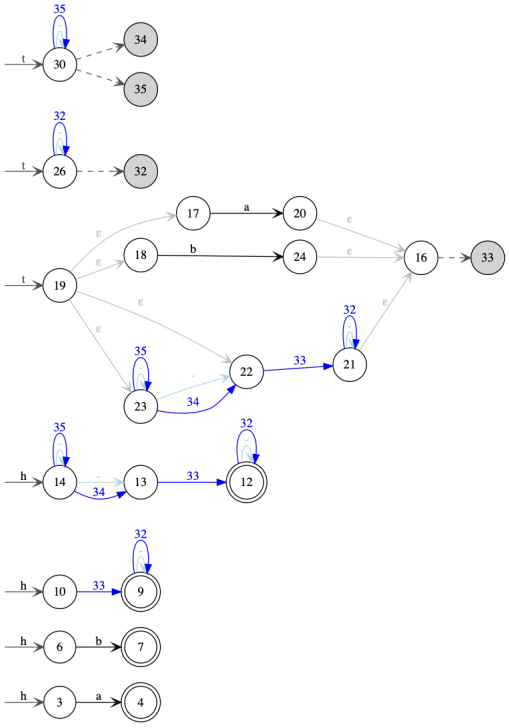

Figure A3. Stepwise hedge automaton from *NWA* for step$(nwa(ch^*(a+b)))$.

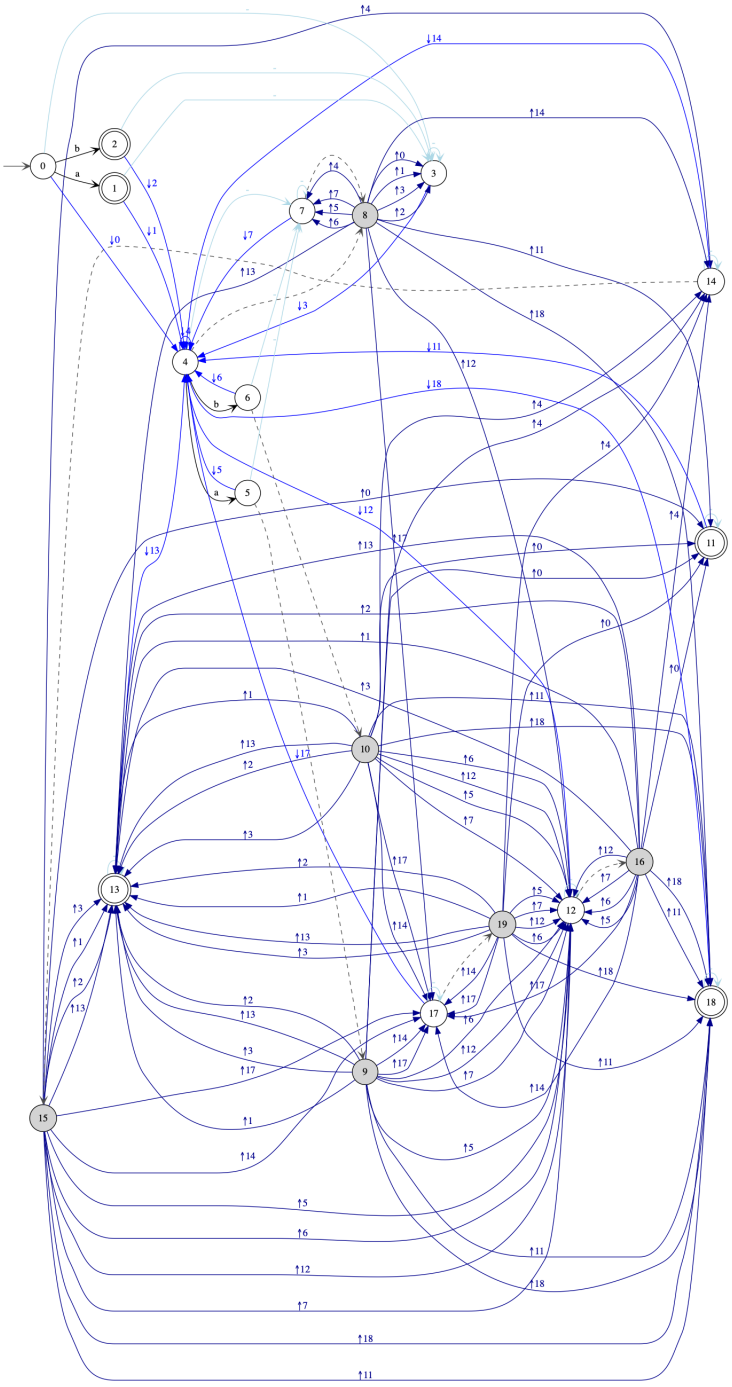

Figure A4. Determinization of *NWA* from stepwise hedge automaton: $det(nwa(\text{step}(ch^*(a+b))))$.

References

1. Mehlhorn, K. Pebbling Moutain Ranges and its Application of DCFL-Recognition. In *Automata, Languages and Programming, Proceedings of the 7th Colloquium, Noordweijkerhout, The Netherlands, 14–18 July 1980*; Lecture Notes in Computer Science; de Bakker, J.W., van Leeuwen, J., Eds.; Springer: Berlin/Heidelberg, Germany, 1980; Volume 85, pp. 422–435, doi:10.1007/3-540-10003-2_89.
2. Von Braunmühl, B.; Verbeek, R. Input Driven Languages are Recognized in log n Space. -North-Holl. Math. Stud., **1985**, *102*, 1–19, doi:10.1016/S0304-0208(08)73072-X.
3. Alur, R. Marrying Words and Trees. In Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Paris, France, 14–16 June 2004 ; ACM-Press: New York, NY, USA, 2007; pp. 233–242.
4. Okhotin, A.; Salomaa, K. Complexity of input-driven pushdown automata. *SIGACT News* **2014**, *45*, 47–67, doi:10.1145/2636805.2636821.

5. Alur, R.; Madhusudan, P. Visibly pushdown languages. In Proceedings of the 36th ACM Symposium on Theory of Computing, Chicago, IL, USA, 13–16 June 2004; ACM-Press: New York, NY, USA, 2004; pp. 202–211.
6. Neumann, A.; Seidl, H. Locating Matches of Tree Patterns in Forests. In Proceedings of the Foundations of Software Technology and Theoretical Computer Science, Chennai, India, 17–19 December 1998; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1998; Volume 1530, pp. 134–145.
7. Gauwin, O.; Niehren, J.; Roos, Y. Streaming Tree Automata. *Inf. Process. Lett.* **2008**, *109*, 13–17, doi:10.1016/j.ipl.2008.08.002.
8. Thatcher, J.W. Characterizing derivation trees of context-free grammars through a generalization of automata theory. *J. Comput. Syst. Sci.* **1967**, *1*, 317–322.
9. Comon, H.; Dauchet, M.; Gilleron, R.; Löding, C.; Jacquemard, F.; Lugiez, D.; Tison, S.; Tommasi, M. Tree Automata Techniques and Applications. 2007. Available online: http://tata.gforge.inria.fr (accessed on 1 February 2021).
10. Hosoya, H.; Pierce, B.C. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.* **2003**, *3*, 117–148, doi:10.1145/767193.767195.
11. Mozafari, B.; Zeng, K.; Zaniolo, C. From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences. *PVLDB* **2010**, *3*, 150–161, doi:10.14778/1920841.1920865.
12. Pitcher, C. Visibly Pushdown Expression Effects for XML Stream Processing. *Program. Lang. Technol. XML* **2005**, *1060*, 1–14.
13. Olteanu, D. SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Trans. Know. Data Eng.* **2007**, *19*, 934–949, doi:10.1109/TKDE.2007.1063.
14. Gauwin, O.; Niehren, J. Streamable Fragments of Forward XPath. In Proceedings of the International Conference on Implementation and Application of Automata, Blois, France, 13–16 July 2011; Lecture Notes in Computer Science; Markhoff, B.B., Caron, P., Champarnaud, J.M., Maurel, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6807, pp. 3–15, doi:10.1007/978-3-642-22256-6_2.
15. Benedikt, M.; Jeffrey, A.; Ley-Wild, R. Stream Firewalling of XML Constraints. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 10–12 June 2008; ACM-Press: New York, NY, USA, 2008; pp. 487–498.
16. Gauwin, O.; Niehren, J.; Tison, S. Earliest Query Answering for Deterministic Nested Word Automata. In Proceedings of the 17th International Symposium on Fundamentals of Computer Theory, Wroclaw, Poland, 2–4 September 2009; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5699, pp. 121–132.
17. Franceschet, M. XPathMark Performance Test. Available online: https://users.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html (accessed on 25 October 2020).
18. Debarbieux, D.; Gauwin, O.; Niehren, J.; Sebastian, T.; Zergaoui, M. Early nested word automata for XPath query answering on XML streams. *Theor. Comput. Sci.* **2015**, *578*, 100–125, doi:10.1016/j.tcs.2015.01.017.
19. Brüggemann-Klein, A. Regular Expressions into Finite Automata. *Theor. Comput. Sci.* **1993**, *120*, 197–213, doi:10.1016/0304-3975(93)90287-4.
20. Brüggemann-Klein, A.; Wood, D. One-Unambiguous Regular Languages. *Inf. Comput.* **1998**, *142*, 182–206.
21. Carme, J.; Niehren, J.; Tommasi, M. Querying Unranked Trees with Stepwise Tree Automata. In Proceedings of the 19th International Conference on Rewriting Techniques and Applications, Paris, France, 26–28 June 2004; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3091, pp. 105–118.
22. Alur, R.; Kumar, V.; Madhusudan, P.; Viswanathan, M. Congruences for Visibly Pushdown Languages. In *Automata, Languages and Programming, Proceedings of the 32nd International Colloquium, Lisbon, Portugal, 11–15 July 2005*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3580, pp. 1102–1114, doi:10.1007/11523468_89.
23. Chervet, P.; Walukiewicz, I. Minimizing Variants of Visibly Pushdown Automata. In *Mathematical Foundations of Computer Science 2007*; Kučera, L., Kučera, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 135–146.
24. Gauwin, O.; Muscholl, A.; Raskin, M. Minimization of visibly pushdown automata is NP-complete. *Log. Methods Comput. Sci.* **2020**, *16*, doi:10.23638/LMCS-16(1:14)2020.
25. Boneva, I.; Niehren, J.; Sakho, M. Nested Regular Expressions Can Be Compiled to Small Deterministic Nested Word Automata. In *Computer Science—Theory and Applications, Proceedings of the 15th International Computer Science Symposium in Russia (CSR 2020), Yekaterinburg, Russia, 29 June–3 July 2020*; Lecture Notes in Computer Science; Fernau, H., Ed.; Springer: Cham, Switzerland, 2020; Volume 12159, pp. 169–183, doi:10.1007/978-3-030-50026-9_12.
26. Gottlob, G.; Koch, C.; Pichler, R. The complexity of XPath query evaluation. In Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, San Diego, CA, USA, 9–12 June 2003; pp. 179–190.
27. Libkin, L.; Martens, W.; Vrgoč, D. Querying Graph Databases with XPath. In Proceedings of the 16th International Conference on Database Theory (ICDT'13), Genoa, Italy, 18–22 March 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 129–140, doi:10.1145/2448496.2448513.
28. Fischer, M.J.; Ladner, R.E. Propositional Dynamic Logic of Regular Programs. *J. Comput. Syst. Sci.* **1979**, *18*, 194–211, doi:10.1016/0022-0000(79)90046-1.
29. Mozafari, B.; Zeng, K.; Zaniolo, C. High-performance complex event processing over XML streams. In Proceedings of the SIGMOD Conference, Scottsdale, AZ, USA, 20–24 May 2012; Candan, K.S., Chen, Y., Snodgrass, R.T., Gravano, L., Fuxman, A., Eds.; ACM: New York, NY, USA, 2012; pp. 253–264, doi:10.1145/2213836.2213866.

30. Grez, A.; Riveros, C.; Ugarte, M. A Formal Framework for Complex Event Processing. In Proceedings of the 22nd International Conference on Database Theory (ICDT 2019), Lisbon, Portugal, 26–28 March 2019; Volume 127, pp. 5:1–5:18, doi:10.4230/LIPIcs.ICDT.2019.5.
31. Bozzelli, L.; Sánchez, C. Visibly Rational Expressions. *Acta Inf.* **2014**, *51*, 25–49, doi:10.1007/s00236-013-0190-6.
32. Gécseg, F.; Steinby, M. *Tree Automata*; Akadémiai Kiadó: Budapest, Hungary, 1984.
33. Scott, D.; de Bakker, J.W. *A Theory of Programs*; IBM: Vienna, Austria, 1969; Unpublished Manuscript.
34. Stockmeyer, L.J.; Meyer, A.R. Word Problems Requiring Exponential Time. In Proceedings of the 5th ACM Symposium on Theory of Computing, Austin, TX, USA, 30 April–2 May 1973; pp. 1–9.
35. Champavère, J.; Gilleron, R.; Lemay, A.; Niehren, J. Efficient Inclusion Checking for Deterministic Tree Automata and XML Schemas. *Inf. Comput.* **2009**, *207*, 1181–1208, doi:10.1016/j.ic.2009.03.003.
36. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Addison Wesley: Reading, MA, USA, 2006.
37. Arnold, A.; Niwiński, D. Complete lattices and fixed-point theorems. In *Rudiments of μ-Calculus*; Studies in Logic and the Foundations of Mathematics; North-Holland: Amsterdam, Netherlands, 2001; Volume 146, Chapter 1, pp. 1–39.
38. Martens, W.; Niehren, J. On the Minimization of XML-Schemas and Tree Automata for Unranked Trees. *J. Comput. Syst. Sci.* **2007**, *73*, 550–583, doi:10.1016/j.jcss.2006.10.021.
39. D'Antoni, L.; Alur, R. Symbolic Visibly Pushdown Automata. In *Computer Aided Verification, Proceedings of the 26th International Conference (CAV, VSL 2014), Vienna, Austria, 18–22 July 2014*; Lecture Notes in Computer Science; Biere, A., Bloem, R., Eds.; Springer: Cham, Switzerland, 2014; Volume 8559, pp. 209–225, doi:10.1007/978-3-319-08867-9_14.