Streaming Tree Automata and XPath

Olivier Gauwin

Université Lille 1 – Sciences et Technologies
Laboratoire d'Informatique Fondamentale de Lille
Institut National de Recherche en Informatique et en Automatique

THÈSE

présentée en première version en vue d'obtenir le grade de Docteur, spécialité Informatique

par

Olivier Gauwin

FLUX XML, REQUÊTES XPATH ET AUTOMATES

(Streaming Tree Automata and XPath)

Thèse soutenue le 28 septembre 2009 devant le jury composé de :

MICHAEL BENEDIKT	Oxford University	Rapporteur
HELMUT SEIDL	Technische Universität München	Rapporteur
HUBERT COMON-LUNDH	ENS Cachan	Examinateur
ARNAUD DURAND	Université Denis Diderot – Paris 7	Examinateur
SEBASTIAN MANETH	University of New South Wales	Examinateur
JOACHIM NIEHREN	INRIA	Directeur
SOPHIE TISON	Université de Lille 1	Co-Directrice

Flux XML, Requêtes XPath et Automates

Olivier Gauwin

Version: June 23, 2010

Remerciements

Cette thèse n'est pas uniquement le fruit de mon travail, c'est également la conséquence de collaborations fécondes et de volontés bienveillantes.

En premier lieu je remercie mes directeurs de thèse, Joachim Niehren et Sophie Tison. Ils m'ont accordé leur confiance dès le premier jour, alors que nous ne nous connaissions pas. Tous deux sont à l'origine de nombre d'idées et de conseils, dont cette thèse est le reflet, et dont leur complémentarité est souvent la source. J'ai apprécié de pouvoir travailler à leur côté durant ces trois ans.

Cette thèse doit également son existence à Pierre Marquis. Par ses enseignements et la supervision de mon stage de DEA avec Sébastien Konieczny, il a su m'initier avec enthousiasme à la science informatique. Je le remercie grandement pour son soutien dans ma recherche d'une thèse.

Anne-Cécile Caron et Yves Roos ont travaillé avec nous sur la partie automates. D'autres travaux ont été initiés avec Arnaud Durand et Marco Kuhlmann. De nombreuses discussions ont également influencé nos travaux, notamment avec Sebastian Maneth et Michael Benedikt. J'ai été très heureux de travailler avec chacun d'entre eux, j'ai beaucoup appris grâce à eux.

Je remercie Michael Benedikt, Helmut Seidl, Hubert Comon-Lundh, Arnaud Durand et Sebastian Maneth d'avoir accepté de faire partie du jury.

Je suis reconnaissant envers l'équipe Mostrare, pour l'état d'esprit stimulant qui y règne. Merci à Rémi Gilleron pour l'accueil et les conseils prodigués. Je salue également les doctorants et post-docs Mostrare. Au-delà de notre amitié, je remercie en particulier Emmanuel Filiot et Sławek Staworko pour avoir répondu à mes nombreuses questions ! Benoît Papegay a commencé l'implémentation de certains algorithmes, je l'en remercie. Je remercie l'INRIA pour avoir financé cette thèse, ainsi que le LIFL et l'université Lille 1 pour les moyens accordés.

Je ne peux bien sûr pas énumérer tout ce qui a éveillé mon intérêt pour la science et l'informatique. J'y inclus mes professeurs de science, que je remercie. Un merci tout particulier à mes parents, pour m'avoir accordé beaucoup de liberté, et plus généralement pour m'avoir fait confiance, soutenu et encouragé. Enfin merci à Laurence pour son soutien, même quand le travail a pris le dessus !

Résumé en Français Ces dernières années, XML est devenu le format standard pour l'échange de données. Les documents XML sont généralement produits à partir de bases de données, durant le traitement de documents, ou au sein d'applications Web. L'échange de données en flux est fréquemment utilisé lors de l'envoi de données conséquentes par le réseau. Ainsi le transfert par flux est adéquat pour de nombreux traitements XML.

Dans cette thèse, nous étudions des algorithmes d'évaluation de requêtes sur des flux XML. Notre objectif est de gérer efficacement la mémoire, afin de pouvoir évaluer des requêtes sur des données volumineuses, tout en utilisant peu de mémoire. Cette tâche s'avère complexe, et nécessite des restrictions importantes sur les langages de requêtes. Nous étudions donc les requêtes définies par des automates déterministes ou par des fragments du standard W3C XPath, plutôt que par des langages plus puissants comme les standards W3C XQuery et XSLT.

Nous définissons tout d'abord les *streaming tree automata* (STAs), qui opèrent sur les arbres d'arité non bornée dans l'ordre du document. Nous prouvons qu'ils sont équivalents aux *nested word automata* et aux *pushdown forest automata*. Nous élaborons ensuite un algorithme d'évaluation au plus tôt, pour les requêtes définies par des STAs déterministes. Bien qu'il ne stocke que les candidats nécessaires, cet algorithme est en temps polynomial à chaque événement du flux, et pour chaque candidat. Par conséquent, nous obtenons des résultats positifs pour l'évaluation en flux des requêtes définies par des STAs déterministes. Nous mesurons une telle adéquation d'un langage de requêtes à une évaluation en flux via un nouveau modèle de machines, appelées *streaming random access machines* (SRAMs), et via une mesure du nombre de candidats simultanément vivants, appelé *concurrence*. Nous montrons également qu'il peut être décidé en temps polynomial si la concurrence d'une requête définie par un STA déterministe est bornée. Notre preuve est basée sur une réduction au problème de la valuation bornée des relations reconnaissables d'arbres.

Concernant le standard W3C XPath, nous montrons que même de petits fragments syntaxiques ne sont pas adaptés à une évaluation en flux, sauf si P=NP. Les difficultés proviennent du non-déterminisme de ce langage, ainsi que du nombre de conjonctions et de disjonctions. Nous définissons des fragments de Forward XPath qui évitent ces problèmes, et prouvons, par compilation vers les STAs déterministes en temps polynomial, qu'ils sont adaptés à une évaluation en flux.

Titre en Français Flux XML, Requêtes XPath et Automates

Mots clés en Français Flux XML, requêtes, arbres, automates, XPath.

Résumé en Anglais During the last years, XML has evolved into the quasi standard format for data exchange. Most typically, XML documents are produced from databases, during document processing, and for Web applications. Streaming is a natural exchange mode, that is frequently used when sending large amounts of data over networks, such as in database driven Web applications. Streaming is thus relevant for many XML processing tasks.

In this thesis, we study streaming algorithms for XML query answering. Our main objective lies in efficient memory management, in order to be able to query huge data collections with low memory consumption. This turns out to be a surprisingly complex task, which requires serious restrictions on the query language. We therefore consider queries defined by deterministic automata or in fragments of the W3C standard language XPath, rather than studying more powerful languages such as the W3C standards XQuery or XSLT.

We first propose *streaming tree automata* (STAs) that operate on unranked trees in streaming order, and prove them equivalent to nested word automata and to pushdown forest automata. We then contribute an earliest query answering algorithm for query defined by deterministic STAs. Even though it succeeds to store only alive answer candidates, it consumes only PTIME per event and candidate. This yields positive streamability results for classes of queries defined by deterministic STAs. The precise streamability notion here relies on a new machine model that we call *streaming random access machines* (SRAMs), and on the number of concurrently alive candidates of a query. We also show that bounded concurrency is decidable in PTIME for queries defined by deterministic STAs. Our proof is by reduction to bounded valuedness of recognizable tree relations.

Concerning the W3C standard query language XPath, we first show that small syntactic fragments are not streamable except if P=NP. The problematic features are non-determinism in combination with nesting of and/or operators. We define fragments of Forward XPath with schema assumptions that avoid these aspects and prove them streamable by PTIME compilation to deterministic STAs.

Titre en Anglais Streaming Tree Automata and XPath

Mots clés en Anglais XML streams, queries, trees, automata, XPath.

Contents

Chapter 1

Introduction

1.1 Background

The XML format, introduced over ten years ago [BPSM+08], has become a *de facto* standard for data exchange. It is now a common language for various communities, from web technologies to document processing and databases. Originating from SGML, XML defines semi-structured documents, modeled by trees. The syntax of an XML document is a well-nested sequence of tags, some of them containing textual content. This differs from relational databases, where the data is stored in tables. With XML appeared schema languages like DTDs (Document Type Definition), XML Schema or Relax NG. A schema is used to define the correct structure of XML documents of some given application.

Consider for instance the XML document in Figure 1.1(a). This represents geospatial data of two cities, and is modeled by the tree in Figure 1.2. A schema for this document is presented in Figure 1.1(b).

The first task for processing XML is to *validate* documents against schemas. This is a requirement for applications that manipulate XML data, in order to check their conformance to the desired schema. The second task is *query answering*, which consists of selecting nodes in an XML document, according to the query. This is a basic step to retrieve information from an XML document. In our example one might want to retrieve triples $(\texttt{name}, \texttt{lat}, \texttt{lon})$. Query answering is a generalization of *filtering*, which requires to determine whether an XML document has a match w.r.t. the query. The third task, and very common use of query answering, is *data transformation*. In the context of XML, this aspect has many applications. Data exchange, for instance, consists of translating a document satisfying a schema, to a document conforming to another schema. In our example, geospatial data can be represented using different schemas by different

```
<geo>
  <point>
    <name>Lille</name>
    <lat>50.63050</lat>
    <lon>3.07063</lon>
  </point>
  <point>
    <name>Hellemmes</name>
    <lat>50.62746</lat>
    <lon>3.10853</lon>
  </point>
</geo>
```

(a) XML document.

$$
\begin{aligned}
\texttt{geo} &\rightarrow \texttt{point}^* \\
\texttt{point} &\rightarrow (\texttt{name}, \texttt{lat}?, \texttt{lon}?) \\
\texttt{name} &\rightarrow \#\text{PCDATA} \\
\texttt{lat} &\rightarrow \#\text{PCDATA} \\
\texttt{lon} &\rightarrow \#\text{PCDATA}
\end{aligned}
$$

(b) DTD schema.

Figure 1.1: XML file containing geospatial data, conforming to a DTD.

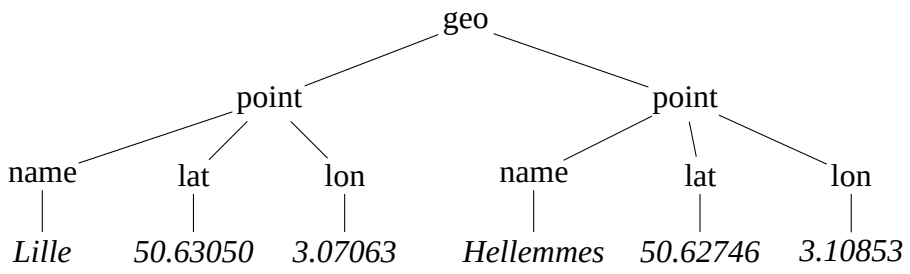


Figure 1.2: The tree representation of the XML file in Figure 1.1(a).

governments or companies, so one might want to export these data into another schema. Data transformation consider all possible transformations from an XML document to another one. Another frequent example is the transformation of XML documents to HTML web pages using XSLT stylesheets.

All these tasks can be performed in several modes. The first mode is the *in-memory evaluation*. Here the whole XML document is loaded into main memory, and then processed. The output is produced only when all the query answers are computed. One drawback of this approach is a significant memory consumption. Another is that often some answers can be produced before the whole set of query answers is computed. An approach to solve the latter deficiency is the *enumeration* of solutions. It consists in outputting, after a preprocessing phase, each solution one at a time, with a reasonable delay between two consecutive answers. Finally, the *streaming* mode imposes stronger restrictions on space usage. In this mode, the XML document is read in only one pass, from the first to the last tag

of the document. The output is also produced in a streaming manner: When an answer is found or a part of the output document is computed, it is immediately output to another device. The objective of a streaming evaluation is to use less memory, by only buffering the required information. Buffering is necessary when the output still depends on the continuation of the stream. The goal is to deal with documents that cannot be loaded into main memory, or to process XML streams coming from the network on the fly.

Several standards have been elaborated for the aforementioned tasks. We already illustrated schema languages at DTDs, defined within the XML recommendation [BPSM$^+$08]. XML Schema [FW04] extends DTDs by adding some features like more precise characterizations of textual content. Moreover, an XML Schema is itself an XML document, unlike DTDs. Relax NG [vdV03] focuses on the description of the structure of valid trees, and delegates the specification of valid textual content to XML Schema.

XPath [CD99] is the standard language for selecting nodes in XML documents. It is based on a description of paths, by series of steps to be followed in order to reach selected nodes. XPath also allows to add filters along these steps. A filter is a Boolean combination of path expressions, and is satisfied if a node matches this combination. It is also possible to test textual content of nodes. The navigational core of XPath 1.0, named CoreXPath 1.0, has been extracted by Gottlob, Koch and Pichler in [GKP05]. XPath is a core query language, used for node selection in many other languages, like XPointer [DMJ01], a standard for selecting fragments of XML documents.

XPath is also used by both popular transformation languages XQuery [BCF$^+$07] and XSLT [Cla99]. XQuery is an imperative language using for-loops in order to select tuples of nodes, that are subsequently inserted in some XML context to produce an output XML document. XSLT is closer to functional programming. An XSLT stylesheet is a set of template rules that are activated on nodes matching XPath expressions.

XProc [WMT09] proposes to combine all these standards using a pipeline language. Whereas XPath, XQuery and XSLT were not designed for streaming evaluation, XProc permits to define parts of the tree where the selection and transformation occur, and thus restricts the inherent difficulties of their streaming evaluation to smaller regions. We will see in this dissertation that other languages, like STX [BBC02], have been designed specifically for streaming evaluation, but no standard has been adopted yet.

Finite word automata [HU79] process words in one pass, to decide their acceptance. Hence, they naturally perform streaming evaluation of words. These objects have been extensively studied, and enjoy interesting relations with logics

and formal languages, as an automaton basically defines a language of words. XML documents are modeled as trees, not words. However, original XML documents are linearizations of these trees: An XML document is a series of tags (an XML stream), and thus a word. Here, tags are well-nested, reflecting the tree structure. Finite word automata are not able to take this nesting relation into account, so we need a more powerful notion of automata to process XML streams.

Tree automata [CDG+07] provide a framework to formally define and study XML tasks. Tree automata also benefit from extensive work, and still relate directly with logics and languages over trees. In particular, they provide an algebraic framework to XML databases, like the relational algebra for relational databases. It has been shown that tree automata could capture all the standard schema languages, and the translation of a schema to a tree automaton is relatively simple [MLM01]. Tree automata were also proposed to define queries in trees [NS02, Koc03, BS04, CNT04]. XPath expressions can be translated into tree automata, but this time the translation is not trivial. Validation (here, named model-checking) and query answering tasks are also studied for tree automata. Transformations are defined by tree transducers. These differ from tree automata by allowing to produce an output while reading an input tree.

1.2 Motivations

In this manuscript, we study the query answering task, using a streaming evaluation, on queries defined by XPath and tree automata. Streaming evaluation is now a major challenge for XPath processing. Michael Kay, the author of the reference XQuery processor Saxon, recently declared [Kay09]:

> The streaming capabilities [of Saxon] are now one of the major reasons people buy the product.

The evaluation of streamed XML documents has been considered for a long time. We illustrate this evaluation mode and related concepts on a query over words on the alphabet $\{a, b\}$. Consider the query that selects positions labeled by a, directly followed by $b \cdot b$. For instance, on the word $a \cdot b \cdot a \cdot a \cdot b \cdot b \cdot b \cdot a \cdot b \cdot b$, this query selects positions 4 and 8, as illustrated in Figure 1.3. All b-positions can immediately be discarded. For a-positions, the selection or rejection cannot be decided immediately. Positions followed by an a (like 3) can be discarded after one step, and those followed by $b \cdot a$ (like 1) after two steps. This query can be answered with a sliding window of length 3, and needs to buffer at most one candidate at a time. We name *delay* the minimal size for the sliding window, and *concurrency* [BYFJ05] the minimal number of simultaneous alive candidates. A candidate is *alive* at a given time point, when there exists a continuation of the

input	a	b	a	a	b	b	b	a	b	b
buffer	1	1	3	4	4			8	8	
output						4				8

Figure 1.3: Streaming evaluation for the selection of a-positions followed by $b \cdot b$.

stream after this time point for which the candidate is selected, and another for which it is rejected. Hence these alive candidates have to be buffered. It is often easy to define small queries with high concurrency, for instance here by allowing that $b \cdot b$ appears after an a, but not immediately after. Schema information can reduce the buffering requirements. For instance suppose that all valid words are such that once three successive b-positions are read, all a-positions are followed by $b \cdot b$. Then all a-positions following three b-positions can be output immediately. For instance here, the position 8 can be safely output at position 8 instead of position 10.

From the beginning, streaming algorithms outperformed other evaluators, but worked on restricted fragments. Many difficulties for streaming evaluation were identified. For the validation task [SV02], a first problem is the recursive nature of XML documents. Processing recursive documents requires storing information about ancestor nodes in a stack. Hence the memory can be bounded by the height of the tree, but cannot be bounded independently for all trees. Query languages like XPath are inherently non-deterministic [PC05], unlike schema languages. For instance XPath allows steps through the *descendant* axis. Starting from one node, this matches all its descendants, thus generating a lot of candidate nodes for the next step. Here, these candidates need sometimes to be buffered, as they might require some information to determine if they satisfy the query (for instance if there is a condition on their next siblings). These difficulties even occur when filtering XML documents using XPath [AF00]. Moreover, XPath allows branching, by allowing filters and conjunctions inside filters. This also often participates in increasing the complexity of algorithms. Transformations impose additional difficulties for streaming [FHM$^+$05, Mic07]. This is typically the case for the operators dealing with positions among selected elements, for instance when looking for the last selected node, or for sorting nodes.

Relative to these blocking aspects, lower memory bounds for these tasks have been established. In the context of query answering, the key notion is the concurrency, as introduced previously. It has been proved [BYFJ05] that the concurrency is a lower memory bound for processing XPath queries, for a fragment of XPath without wildcards. This raises a challenging issue: can we reach this bound? This question can be decomposed into several variants. First, can this result be gener-

alized to larger query classes? It would also be interesting to know whether the bound is tight, i.e. whether there exist algorithms which memory consumption is tight from this lower bound. What is the cost in time for reaching such bounds, i.e. do these algorithms require a lot of computation, in order to decide the selection or rejection of candidates? How does this cost vary from a query class to another? In other words, are there query classes for which efficient algorithms exist? Can we characterize such query classes by some property? Can queries with unbounded concurrency be tractable for streaming? Which queries require low buffering (even though unbounded)? These questions motivate a notion finer than concurrency: the *streamability* of a query, i.e. a measure of appropriateness to streaming evaluation. The concurrency draws a first frontier, between queries having bounded concurrency (and thus using bounded memory on every document of bounded-depth) and the remaining one. But the questions above call for a more fine-grained notion of streamability.

Beyond filtering and monadic node-selection queries, we study n-ary queries, for $n \geq 0$. These are queries selecting n-tuples of nodes in trees. The case $n = 0$ corresponds to Boolean queries that can only distinguish trees selecting the empty tuple, and hence define tree languages. They are used to filter trees satisfying some constraints. For $n = 1$, we obtain monadic queries, that select, for each tree, a set of nodes in this tree. The selection of n-tuples of nodes is a core operation in transformation languages. For XPath 2.0 and XQuery, this operation is done through nested for-loops called FLOWR expressions. XPath 1.0 only defines monadic queries. By introducing variables, we allow XPath 1.0 to define n-ary queries. Compared to FLOWR expressions, this permits more flexibility in terms of evaluation, and might complicate the task of our algorithms. FLOWR expressions are more low-level instructions, that might help the developer to define queries suitable to streaming, or not. For queries by automata, n-ary queries are defined by languages of annotated trees.

Reaching the memory lower bound is very time consuming. Benedikt et al. [BJLW08] show for instance that for XPath used with DTDs, rejecting failed candidates at the earliest time point with an algorithm built in polynomial time in the size of query, with per-event polynomial time in the size of the query, is equivalent to PTIME = PSPACE.

Berlea [Ber06, Ber07] study *regular tree queries*, defined by tree grammars. For this query class, Berlea proposes an algorithm based on tree automata, that uses optimal memory management (in terms of stored candidates), while enjoying PTIME per-event and per-candidate space and time. However, this query class assumes an infinite alphabet, even for labels. This differs from the XML format, where only textual contents (i.e., data values) are unrestricted. The fact that the alphabet is infinite indeed simplifies earliest selection or rejection of candi-

dates tremendously. In particular, this query language is not closed by complement. The algorithm can however be used for answering positive XPath expressions in PTIME, when assuming a bound on the branching width of XPath expressions. Moreover, this algorithm efficiently processes queries defined by non-deterministic automata.

Some algorithms were proposed for the streaming evaluation of XPath. For downward axes, we can mention the work by Bar-Yossef et al. [BYFJ05, BYFJ07], Ramanan [Ram05, Ram09], and Gou and Chirkova [GC07a]. Algorithms by Barton et al. [BCG$^+$03] and Wu and Theodoratos [WT08] allow both upward and downward axes. Olteanu et al. [OMFB02, OKB03, Olt07b] prove that Forward XPath, the fragment of XPath 1.0 where all axes respecting the document order are allowed, is as expressive as CoreXPath 1.0. They propose *SPEX*, an efficient algorithm based on transducers networks, that evaluates all Forward XPath expressions. Nizar and Kumar [NK08] define an algorithm for Forward XPath expressions where no negation occurs. Recently, they extend their framework [NK09] to allow backward axes. Benedikt and Jeffrey [BJ07] study logics equivalent to CoreXPath 1.0, and their appropriateness for streaming. They identify fragments using backward and downward modalities without negation, such that the selection of a node can be decided when opening (resp. closing) it. They show that for these fragments polynomial per-event space and time algorithms exist. Benedikt et al. [BJLW08] study the filtering of XML streams against XPath constraints, and introduce a heuristic for the earliest detection of violated constraints. All these algorithms for the evaluation of XPath over XML streams do not achieve optimal memory management, and store useless candidates (or partial matches) in some cases. Ley and Benedikt et al. [LB09] study whether there exist extensions of XPath being as expressive as the first-order logic, and using only forward axes. They prove that the first-order complete extensions used when all axes are allowed do not suffice when restricted to forward axes.

Other lower bounds were also established, in addition to concurrency. Bar-Yossef et al. [BYFJ04, BYFJ07] establish three lower bounds, for some fragments of XPath. The first one is the *query frontier size*, i.e. the maximal number of siblings of all ancestors of a node, in the tree representation of the query. The second one is the *recursion depth* of the document, which corresponds to the maximal number of ancestors with the same label. The third one is the logarithmic value of the *depth* of the tree. Grohe, Koch and Schweikardt [GKS07], while studying Turing machines modeling stream processing with multiple scans, establish that for CoreXPath 1.0, the *depth* of the tree is a lower bound. A more complete state of the art is provided in Section 1.4.

1.3 Contributions

We now present our contributions. Throughout this manuscript, we consider n-ary queries, i.e. queries that select n-tuples of nodes, instead of simple nodes, as allowed by XPath 2.0. Moreover, we always try to take advantage of schemas to make stream processing more efficient, as schemas are often available in concrete applications.

Streamability We start by defining a computational model for streaming query answering: the *Streaming Random Access Machines* (SRAMs). We then introduce our notion of *streamability*. We have seen that such a notion is lacking in the current state of the art. In particular, the absence of such formal definitions leads to a number of errors in the space complexity analysis of many papers. Roughly speaking, for a natural number m or $m = \infty$, a query is m-streamable if it can be computed using polynomial space and time for all trees for which the concurrency of the query is less than m. This sets up a hierarchy of query classes. m-streamability with a high value of m is desirable, and means that input trees with concurrency lower than m can be efficiently processed. ∞-streamable queries are called *streamable* queries, and always use polynomial per-event time and space, independently of the concurrency. We study the relations between query classes that are ∞-streamable, and query classes that are m-streamable for all $m \in \mathbb{N}_0$. Query classes being m-streamable for all $m \in \mathbb{N}_0$ must have polynomially bounded concurrency in order to be ∞-streamable (for monadic queries). We study the hardness of deciding whether a query class has bounded (resp. polynomially bounded) concurrency. For Forward XPath, these problems are coNP-hard. We show that being 1-streamable implies a PTIME universality test on the class of queries, whenever this class verifies some properties. As universality for Forward XPath queries is coNP-hard, Forward XPath is not 1-streamable, and thus not m-streamable for all $m \in \mathbb{N} \cup \{\infty\}$.

Streaming Tree Automata We define *Streaming Tree Automata* (STAs) as a notion of tree automata that performs pre-order traversals of trees. This corresponds to streaming traversals of XML documents. STAs are a reformulation of nested word automata [Alu07] that operate directly on trees instead of nested words. We show the equivalence between STAs and other automata notions that traverse trees (or encodings of trees) in pre-order: pushdown forest automata [NS98], visibly pushdown automata [AM04] and nested word automata. We also exhibit back and forth translations between STAs and standard (bottom-up and top-down) tree automata. Queries defined by deterministic STAs (dSTAs) are m-streamable for all $m \geq 0$, when bounding the depth of trees. We proved it by elaborating an

earliest query answering algorithm.

Earliest Query Answering for Streaming Tree Automata Earliest query answering (EQA) algorithms have the property of writing answers at the earliest point onto the output stream. In other words, each answer is output once there is enough information to ensure that this answer will be selected on any continuation of the stream. Symmetrically, all rejected candidates are discarded when they fail in all continuations (a property named *fast-fail* in [BJLW08]). These notions originate from the work of Bar-Yossef et al. [BYFJ05] and Berlea [Ber06]. While Bar-Yossef derived lower memory bounds for streaming, we prove time lower bounds, by studying decision problems inherent to EQA algorithms.

The property of being earliest is a requirement for algorithms buffering only alive candidates: not being earliest means that at some point, a candidate is stored while it does not have to. However, being earliest is often computationally complex. For XPath queries, we show that it is coNP-hard to decide whether a prefix of the stream ensures selection of a given candidate. For queries defined by dSTAs, this task becomes tractable, and our earliest query answering algorithm runs in PTIME (for fixed arity n). This proves that dSTAs are a robust formalism for defining streamable queries. Our working hypothesis is that every class of streamable queries can be translated in PTIME to dSTAs. This is for instance the case for the streamable fragment of XPath defined below, for which we provide such a translation, hence proving its streamability.

XPath We then study the streamability of XPath in more details. We identify a hierarchy of fragments, named k-Downward XPath (with $k \in \mathbb{N}$), that are m-streamable for all $m \geq 0$. Here, the key property is that in k-Downward XPath, the number of correct matches of a branch of the expression w.r.t. the tree is at most one at any time point. In order to ensure this property, we combine syntactic restrictions (on the query) with semantic restrictions (on the schema). k-Downward XPath is a rich fragment, in that it allows negation, branching (and thus disjunction), and downward axes (child and descendant). We provide an effective PTIME translation of k-Downward XPath expressions to dSTAs. Hence we can reuse all our algorithms for dSTAs on k-Downward XPath, and in particular the EQA algorithm.

Bounding Concurrency and Delay Finally we prove that for queries defined by dSTAs, it can be decided in PTIME whether a query has bounded delay and/or bounded concurrency. The *delay* is the maximal number of events between reading a selected node (or tuple of nodes in the n-ary case) and the earliest point where its selection can be decided. Delay and concurrency are key streamability

measures: delay is related to the quality of service, while concurrency is a measure of buffering requirements. To obtain these decidability properties, we use and extend results on recognizable relations over trees, that were already studied in the ranked case [Tis90, CDG$^+$07] and also the unranked case [BL02, BLN07]. These are relations over trees that can be recognized by an automaton, modulo an encoding of tree relations into tree languages. We prove that the bounded and k-bounded valuedness of binary recognizable relations can be decided in PTIME, by reduction to bounded valuedness of tree transducers [Sei92] and k-bounded ambiguity of tree automata. This also allows us to decide in PTIME whether a query has a k-bounded delay and/or a k-bounded concurrency, for fixed k and fixed arity n.

1.4 State of the Art

This section surveys the recent work on stream processing, in the context of XML databases. For a survey on streaming more generally, we refer the reader to [Mut05]. We start by enumerating several models for stream processing. We present known lower bounds for XML stream processing, and then exhibit upper bounds by listing known algorithms for processing XML streams.

Models for Stream Processing

Turing machines with multiple tapes, and restrictions on the direction of head moves or on the number of head reversals are studied for a long time [HU69]. These restrictions define new classes of computable languages. Gurevitch, Leinders and Van den Bussche [GLdB07] consider stream queries as particular functions from stream to stream. They study which functions mapping an input stream to an output stream are computable, and in particular which of them are computable with bounded memory. Babcock et al. [BBD$^+$02] previously surveyed some common problems for stream processing, and how they are handled in existing data stream management systems (DSMS).

Grohe, Koch and Schweikardt [GKS07, Sch07a] investigate Turing machines with one external tape where the input is read (and writing is allowed under some conditions), an output write-only tape, and internal tapes without restrictions. They define a hierarchy of machines: machines allowing $k + 1$ head reversals on the input tape are strictly more expressive than machines allowing k reversals. Schweikardt [Sch07a] surveys generalizations of stream processing models, where data can be stored in external (and thus slower) devices (on this precise topic, see also the survey by Vitter [Vit01]).

The expressiveness of query languages over trees is often established w.r.t.

two yardstick logics: the first-order logic (FO) and the monadic second-order logic (MSO), with predicates describing tree structures. Marx shows in [MdR05] that CoreXPath 1.0 is strictly less expressive than FO. Queries defined by tree automata are exactly MSO-definable queries, by the standard equivalence between tree automata and MSO logic established by Doner [Don70] and Thatcher and Wright [TW68]. Ley and Benedikt [LB09] study whether there exists a first-order complete logic using only forward axes, i.e. axes that respect document order. For this purpose, they adapt and combine two modal logics. The first one is the Linear Temporal Logic (LTL) and the second one the Computational Tree Logic (CTL*), which is a temporal logic with branching. They show that using LTL for vertical path expressions together with CTL* for horizontal and downward moves leads to a first-order complete logic. However this logic uses backward moves. Unfortunately, the first-order completeness is lost when restricting to only forward moves, or when restricting the nesting depth of *until* operators in LTL.

We also note that streaming query answering is a particular case of the view maintenance problem (i.e. maintaining the answer set after updates of the document), where only insertions of nodes are allowed [SI84, GMS93, BGMM09].

Lower Bounds

In [GKS07], Grohe, Koch and Schweikardt apply techniques from communication complexity to prove lower bounds. They show that, as a consequence, for *filtering* CoreXPath 1.0 queries the *depth* of the input tree t is a lower memory bound, i.e. there is no streaming algorithm using less than $o(\mathit{depth}(t))$ buffering space for input trees t.

Communication complexity [Yao79, KN97] is a powerful tool for proving lower bounds. It characterizes the minimal amount of information needed to compute a function by two agents, each of them knowing a part of the input.

In [BYFJ04, BYFJ07], Bar-Yossef et al. use this technique to exhibit other lower bounds on a fragment of XPath named Redundancy-free XPath. The bounds apply even for filtering. A key property of Redundancy-free XPath is that a node of the tree cannot match several distinct query nodes. These bounds are formulated w.r.t. the *instance data* complexity, i.e. in terms of properties of each query and document to be evaluated, as opposed to the worst-case complexity. A first memory lower bound on Redundancy-free XPath is the *query frontier size*. When a query Q is represented as a tree, the frontier size at a node of this tree is the number of siblings of this nodes, and its ancestors' siblings. The query frontier size of Q is the largest frontier over all nodes of Q. The second lower bound is the document *recursion depth*. The recursion depth of a tree t w.r.t. a query Q is the maximal number of nested nodes matching a same node in Q. The last lower bound is $\log(d)$, where d is the depth of the document t. This latter lower bound is

smaller than the recent bound proved by Grohe et al. for filtering CoreXPath 1.0 [GKS07] mentioned above.

In a subsequent work [BYFJ05], Bar-Yossef et al. prove that the *concurrency* of a query is a memory lower bound, on Star Free XPath, the fragment of Core-XPath 1.0 with only downward axes (*self*, *ch* and ch^*) and without wildcards. More precisely, the concurrency is proved to be a lower memory bound for the worst-case complexity. For instance data complexity, it is proved that there exists a document, almost similar to the original one, that requires the concurrency in terms of space.

Benedikt et al. [BJLW08] study the feasability of *fast-fail* filtering for XPath with DTDs. Fast-fail means that it must be decided at the earliest time point whether the stream is rejected by a given XPath filter. They prove that PTIME $=$ PSPACE is equivalent to having a PTIME algorithm compiling XPath filters to fast-fail algorithms using polynomial per-event time complexity (in the size of the XPath filter and DTD). Moreover, Benedikt and Jeffrey [BJ07] prove that there is no subexponential function f such that all positive CoreXPath 1.0 filters Q can be computed by algorithms using $f(|Q|, |\Sigma|)$ total space, on bounded-depth trees, even when fast-fail is not required.

Lower bounds were also established in more general frameworks. Arasu et al. prove some lower bounds for the streaming evaluation of conjunctive queries, with multiple input tapes [ABB$^+$04], and more general streamed data. The aforementioned work by Grohe et al. [GKS07] contains additional results when reversals on the input tape are allowed. Recently, Schweikardt extends this framework by allowing multiple input tapes [Sch09]. Communication complexity was already used to prove lower bounds for some streaming problems on relational databases, for instance by Henzinger et al. [HRR99].

Validation

We now survey upper bounds for XML streams processing, by mentioning known algorithms. The easiest task when processing XML documents is the validation, i.e. determine whether a document conforms to a given schema. This problem was first addressed by Segoufin and Vianu [SV02]. In this paper, the authors are looking for DTDs for which the validation can be done with bounded memory. This is not the case for all DTDs. They prove that it is sufficient for the DTD to be non-recursive, or to be fully recursive. A DTDs is fully recursive if all labels leading to recursive labels are mutually recursive. This property can be checked in EXPTIME for DTDs, and in PTIME for deterministic DTDs. However, this condition is not proved to be necessary, and the problem is still open. Some progress was obtained by Segoufin and Sirangelo in [SS07], where the approach is based on finite state automata checking only local properties of trees. For non-recursive

DTDs, Chitic and Rosu [CR04] prove an exponential lower and upper bound for computing the equivalent minimal deterministic automaton (this automaton also checks that the document is well-nested). We note that a precise characterization of schemas that can be validated with constant space is known for another stream encoding, where the labels are not given in closing tags [BLS06].

Chitic and Rosu [CR04] also relax the constant-memory requirement by allowing the size to be logarithmic in the size of the input stream. They present syntactic restrictions on recursive DTDs, so that they can be validated with logarithmic space in the input stream size.

A weaker requirement for validating XML streams is to bound the space by the depth of the input tree. In [SV02], Segoufin and Vianu already show that every EDTD can be translated into a deterministic pushdown automaton, whose stack usage is bounded by the depth of the input tree. Moreover, they show that any DTD can be compiled into an equivalent EDTD of quadratic size, for which the validation is done with bounded memory. In [GKPS05], Gottlob et al. show that the validation problem for XML streams varies from LOGSPACE to LOGCFL, depending on the schema language and representation.

For the more specific problem of typing, Martens et al. [MNS05] prove that typing each node of an XML document at its opening event w.r.t. a restrained competition EDTD can be done in streaming mode. Such a construction, using visibly pushdown automata, is for instance provided by Kumar et al. in [KMV07][1]. An alternative algorithm, avoiding the static construction of the whole automaton, is proposed by Schewe et al. in [STW08]. Martens et al. also prove that non-restrained competition EDTDs cannot be typed in a streaming manner. Martens et al. [MNSB06a] study the precise expressiveness of XML Schema, and propose to replace a constraint of XML Schema (Element Declarations Consistent) by the one-pass pre-order typing requirement. Typing is also sometimes used as a pre-processing phase for further querying, as proposed for instance by Russell et al. in [RNC03].

Filtering

Filtering XML documents is similar to validation in that it defines valid trees, but differs by the specification language. Whereas validation relies on schema languages, filtering trees w.r.t. a given XPath expression consists in selecting trees in which this XPath expression selects at least one node. Altinel and Franklin, in a seminal work [AF00], introduce the framework of *selective dissemination of information*, where many XML documents have to be filtered w.r.t. many XPath

[1] We show in Chapter 4 how to translate a DTD into a Streaming Tree Automaton, which is a similar construction.

expressions, for publish/subscribe systems. They propose an algorithm called *XFilter* for this purpose, based on a translation of non-branching XPath expressions to automata, that are then combined and indexed for efficient filtering. A number of alternative algorithms were proposed, like YFilter [DFFT02, DRF04], which improve XFilter by another method for combining automata, and XTrie [CFGR02], that proposes a better data structure.

In [GMOS03, GGM+04], Green et al. propose XML*TK*, a system based on the translation of XPath queries to a finite word automaton. Hence the events can be processed with constant time. However the automaton has first to be determinized, causing a blow-up in the filter size. This can be sometimes avoided by building the automaton on demand, but the worst case remains the same. The automaton is just an intermediate representation of the query, and the algorithm uses it together with a stack (bounded by the depth of the tree) during the execution. In [GS03b], Gupta and Suciu define XPush machines, that directly use deterministic pushdown automata.

All these systems have either strong restrictions on XPath expressions (no predicates, or predicates that do not require look-aheads) or lead to exponential algorithms. Bar-Yossef et al. [BYFJ04, BYFJ07] prove the tightness of their lower bounds by an algorithm using $\tilde{O}(|Q| \cdot r \cdot log(d))$ in space, where $\tilde{O}$ removes logarithmic factors, and d (resp. r) is the depth (resp. recursive depth) of t.

Benedikt and Jeffrey [BJ07] investigate filtering algorithms with space (and per-event time) independent of the input stream, and polynomial in the filter. They show that this holds for two classes of queries. The first one is a fragment of positive CoreXPath 1.0 (using backward, i.e. up and left, axes), and the second one a fragment of Conditional XPath, also using backward axis. The backward restriction does not weaken the expressiveness: in both fragments, any non-backward query can be rewritten to a backward one. The techniques are similar to the ones used by Olteanu for SPEX [Olt07b] (as explained later for monadic queries): a translation of queries into transducers networks, and a proof that the restriction on axis does not change the expressiveness.

Benedikt et al. [BJLW08] study the problem of firewalling XML streams under XPath constraints. This is similar to filtering, except that the goal here is to detect XML messages violating XPath constraints, and reject them as soon as possible. We already discussed about the hardness of this *fast-fail* feature. The authors propose however a tractable solution, by using binary decision diagrams (BDDs) for implementing automata (here the trees are of bounded depth), by using a heuristic for fast-fail, and by restricting XPath queries (no wildcards, no rightward moves, and no data joins). When compared to transducers networks, BDDs offer better static analysis opportunities.

Query Answering

XPath with Downward Axes *TwigM* [CDZ06] consider monadic XPath queries using only downward (*ch* and *ch**) axes. TwigM focuses on an efficient data structure for storing pattern matches, and deals with positive downward XPath expressions, i.e. tree patterns. *StreamTX* [HJHL08] aims at adapting the TwigStack algorithm to stream processing of tree patterns, while allowing selection of tuples of nodes, instead of nodes. *XSQ* [PC05] does neither allow negation, but includes aggregators and data values comparisons. The core of XSQ is a hierarchy of pushdown transducers, with additional buffers. Chen et al. [CLT$^+$08] consider a streaming evaluation of *generalized tree patterns*, that consist in tree patterns augmented with the *for-let-return* (FLOWR) expressions of XQuery.

Ramanan [Ram05, Ram09] proposes an algorithm that allows negation and downward axes. Its complexity is $O((\mathit{depth}(t) + \mathit{concur}_Q(t)) \cdot |Q|)$ in space and $O(|t| \cdot |Q| \cdot \mathit{depth}(t))$ in time, in the worst case. An extension with backward axes *prec* and $(ns^{-1})^*$ is also presented in [Ram09]. Gou and Chirkova [GC07a] provide another algorithm for downward XPath, with linear combined complexity $O(|Q| \cdot |t|)$. This paper however seems too optimistic by asserting optimal buffering. We will see later on that this requires non-polynomial time (unless PTIME $=$ NP) on downward XPath. Bar-Yossef et al. [BYFJ05] prove that the concurrency lower bound is tight, by an algorithm that uses, on non-recursive documents t, $O(\mathit{concur}_Q(t) + |Q| \cdot (log(|Q|) + log(|t|)))$ space and $\tilde{O}(|Q| \cdot |t|)$ time, where $\tilde{O}$ removes logarithmic factors.

XPath with Downward and Upward Axes Beyond downward axis, some algorithms were proposed for dealing with parent (ch^{-1}) and ancestor ($(ch^{-1})^+$) axis, together with downward axis. This increases the difficulty, as the algorithm has to process the query in a bottom-up way, by guessing whether descendant nodes will further match. This implies high buffering cost. *Xaos* [BCG$^+$03] allows both downward and upward axes in XPath, and starts by converting upward axes to downward axes. One drawback of Xaos is that answers are output only when the input stream ends. Wu and Theodoratos [WT08] propose an alternate algorithm, called *PSX*, for the same set of queries, represented as *partial tree-pattern queries*. By using a stack-based technique to encode matches, they outperform Xaos. *TurboXPath* [JFB05] is an XML stream processor evaluating XPath expressions with downward and upward axis, together with a restricted form of *for-let-where* (FLOWR in XQuery) expressions. Hence, TurboXPath returns tuples of nodes instead of nodes, i.e. processes n-ary queries.

Forward XPath and Variants Forward XPath is the fragment of XPath using only forward axes, i.e. downward axes, plus next-sibling, its transitive closure,

and the axis *foll* that moves to all nodes following the next sibling of the current node in document order. As shown by Olteanu et al., Forward XPath is very expressive, as adding backward axes to Forward XPath does not change its expressiveness [OMFB02, Olt07a]. However translating an XPath expression with backward axes to a Forward XPath expression can imply an exponential blow-up in the size of the expression. Ley and Benedikt [LB09] prove that Conditional XPath does not enjoy this property, i.e., Conditional XPath with only forward axes is not as expressive as Conditional XPath.

SPEX [OMFB02, OKB03, Olt07b] uses a transducers network as query evaluator. Each element of the XPath expression (label test, axis, etc) is translated into a simple transducers, equipped with a stack. Transducers are linked according to the query structure. For instance a step *ch*::a is translated into two transducers, one for *ch* and one for a. The output of the *ch*-transducers conveys an XML stream, that is the input of the a-transducer. This way, a DAG of transducers is built.

Nizar and Kumar [NK08] propose an algorithm for an extension of monadic tree patterns, where axes *foll* and ns^* are allowed. Hence this algorithm defines monadic queries where the negation is not allowed. The complexity of this algorithm is not given, and only experimentally studied. Recently, the authors also investigate the streaming evaluation of monadic tree patterns with additional backward axes *prec* and ns^{-1} [NK09].

Desai [Des01] defines Sequential XPath, a fragment where only forward axes are allowed in path expressions (outside filters), and only backward axes are allowed in filters. In this fragment, selection of a node can be decided at opening time, and thus no buffering of candidates is required. The memory consumption only depends on the depths of the input tree and the Sequential XPath expression.

CoreXPath 1.0 Clark [Cla08] proposes a translation of CoreXPath 1.0 expressions (interpreted as binary queries) to visibly pushdown automata, inspired from the standard translation of MSO formulas to equivalent automata [Don70, TW68, CDG$^+$07]. All axes are allowed. The resulting visibly pushdown automata are non-deterministic, and recognize trees annotated with two variables (corresponding to the canonical language of the queries in our framework). The complexity is non-elementary in the size of the expression, i.e., it cannot be bounded by a tower of exponentials of fixed height. It becomes polynomial when negations are forbidden and the branching width (i.e., the number of leaves in the tree representation of the expression) is bounded. Such translations permit to reuse algorithms designed for queries by automata, with XPath expressions.

Queries by Automata One of the first models for evaluating queries in streaming mode on semi-structured documents was proposed by Neumann and Seidl [NS98, Neu00]. They define monadic queries on forests, i.e. sequences of trees (called *hedges* in this manuscript). Queries are defined by means of *forest grammars*, rephrased as a patterns language of contexts. The selection is made through a special tree variable, and the query selects nodes of the forest where this tree variable can be used. In terms of expressiveness, this corresponds to forest regular languages [Tak75], and regular tree languages when restricted to trees. Neumann and Seidl introduce *pushdown forest automata* in order to evaluate these queries while parsing the XML document, and thus in a streaming way. The links between *pushdown forest automata* and the model of STAs we use in this manuscript are studied in Chapter 4, and show that the models are similar. In particular we provide translations between these models, that allow to change the automaton model behind streaming algorithms.

In the general setting, the evaluation of queries defined by forest grammars using pushdown forest automata is done in two traversals of the tree (left-to-right and then right-to-left). By adding constraints to the grammar, they define *right-ignoring* grammars. These grammars have the property that when traversing the document in streaming order, it can be decided whether a node is selected at closing time. Berlea and Seidl present an extension of this model for n-ary queries [BS04]. They keep the same framework: Queries are defined by grammars, and evaluated using pushdown forest automata.

Berlea [Ber06, Ber07] extends these results to an algorithm that evaluates, in one pre-order traversal of the tree, queries defined by forest grammars (named *regular tree grammars* in the paper). His algorithm is also based on pushdown forest automata, and achieves close to optimal memory usage. As the alphabet of labels is infinite, it is easier to decide whether a state of the automaton will accept all possible continuations. However, the XML format restricts labels to a finite set, and the algorithm is less efficient on finite alphabets. For instance, consider the XPath expression $//a[\mathit{not}(\mathit{not}(a)\ \mathit{and}\ \mathit{not}(b))]$, that selects all a-nodes whose children are all labeled by a or b. If the alphabet is known to be $\Sigma = \{a, b\}$ then all a-nodes can be selected immediately. This cannot be done by the algorithm proposed in [Ber06], and this algorithm will take a decision for the selection of an a-node when closing it. For infinite alphabets, the difference is that a wildcard test is always satisfied, and not a finite union of label tests.

Some results similar to the aforementioned work by Neumann and Seidl [NS98] were established by Kumar et al. in [KMV07], who use *visibly pushdown automata* instead of pushdown forest automata. In particular, the authors exhibit the logic Pre-MSO, corresponding to MSO-definable queries for which the selection of a node only depends on its prefix tree. They show that queries defined by Pre-MSO formulas can be efficiently processed by visibly pushdown automata,

using constant per-event time, and memory in $O(\mathit{depth}(t))$, where t is the input tree. However, the translation of a Pre-MSO formula to such an automaton is non-elementary [AM04, AM06]. In a follow-up work [MV08], Madhusudan and Viswanathan show that queries defined by visibly pushdown automata can be efficiently processed. However, the authors hide a crucial point, as they suppose that the states of the automaton already have enough information to decide whether they are universal, i.e. whether the residual language they accept is any correct continuation of the stream. We propose in Chapter 5 a construction for obtaining such a property for all states, and prove that an exponential time is required for this.

Transformations

Beyond node-selection queries, the streaming evaluation mode is also used for transforming XML documents. Several XQuery processors were proposed for XML streams. Ludäscher et al. [LMP02] translate XQuery expressions into a network of XML Stream Machines (*XSM*) that take XML streams as inputs, and output other XML streams. Finally, the network is compiled into a C program. Koch et al. propose *FluXQuery* [KSSS04a], an XQuery processor based on the intermediate language FluX. FluX adds a *process-stream* instruction to XQuery, that makes the use of buffers more explicit. In [KSSS04b], the authors show how schema information can be used to improve the translation to FluX programs. *GCX* [SSK07] reduces the amount of data to be buffered by purging them using a garbage collector. This one is based on static and dynamic analysis of the query. Fernández et al. [FMSS07] analyze which parts of queries can be evaluated in a streaming manner. They build query execution plans that combine some parts of the query in streaming mode, and other parts using common in-memory techniques. Wei et al. [WRML08] try to reduce space consumption when XML documents are recursive. *Tukwila* [IHW02] is an XQuery processor that evaluates numerous XQuery expressions on an XML stream. The core of Tukwila is based on a stack and a meta-automaton that enables and disables deterministic finite automata that represent linear path expressions of queries.

XSLT is another transformation language based on templates that are activated by XPath expressions defining their execution context. Hence this language is suited to be modeled by transducers. Dvoráková and Rovan [DR07] propose to adapt this idea to a streaming evaluation.

Other transformation languages for XML have been specifically conceived for streaming purpose. *STX* [BBC02] is an event-driven programming language. It is based on templates that specify which operations should be done on the data matching the template pattern. In [KS07], Koch and Scherzinger propose to add attribution functions to the rules of DTDs. These functions are executed

while the document is parsed, and can produce an output. This way, these DTDs (named XML Stream Attribute Grammars) define transformations. By requiring a strong notion of one-unambiguity for the regular expressions, the document can be parsed with a look-ahead of 1. Hence the memory consumption can be bounded (when assuming a bound on the depth of trees). A previous version of this framework named *TransformX* can be found in [SK05]. Frisch defines *XStream* [FN07], a functional programming language that efficiently performs XML transformations. The execution plan of *XStream* is elaborated dynamically, to take advantage of the execution context. Frisch [Fri04] also proposes an efficient implementation of pattern-matching in *CDuce* [BCF03], using tree automata. These operate in document order, and thus the pattern-matching algorithm deals with XML streams. *XTiSP* [Nak04] is another transformation language for XML streams. XTiSP uses as underlying model macro tree transducers, i.e. tree transducers augmented with an accumulator.

1.5 Outline

Chapter 2 introduces the basic objects that we study in this manuscript: unranked trees, schema languages, and queries. It also provides a state of the art about query evaluation.
Chapter 3 defines our model of streaming, and the state of the art for streaming query answering. We introduce the notion of m-streamability, and show that large query classes are not streamable.
Chapter 4 is devoted to Streaming Tree Automata, a model of tree automata adapted to streaming. Beyond the definition, we explicit the link with other existing models of tree automata.
Chapter 5 studies the streamability of deterministic Streaming Tree Automata (dSTAs). For this purpose, we propose an earliest query answering algorithm for queries defined by dSTAs.
Chapter 6 exhibits streamable fragments of XPath. This is mainly proved by a PTIME translation of XPath queries of these fragments to dSTAs.
Chapter 7 proves that deciding whether a query defined by dSTAs has a bounded (resp. k-bounded) delay and concurrency can be done in polynomial time, for a fixed k and a fixed arity n.

1.6 Author's Publications

Streaming Tree Automata Our model of Streaming Tree Automata was established with the collaboration of Anne-Cécile Caron and Yves Roos, and presented

in [GNR08]. Chapter 4 contains the results of this paper, with extra back and forth translations between STAs and standard tree automata.

Earliest Query Answering The definition of earliest query answering and relative hardness results described in Chapter 5 were presented in [GNT09b], and a preliminary version in [GCNT08]. This is also a joint work with Anne-Cécile Caron and Yves Roos.

Bounded Concurrency and Delay The PTIME decision procedures for deciding bounded delay and concurrency of queries defined by dSTAs were presented in [GNT09a]. Chapter 7 contains the results of this paper, with additional improvements. The main improvement is the procedure for deciding the k-bounded delay and concurrency in PTIME for a fixed k (it is in NP in the paper). We also prove that when k is variable, the problem becomes EXPTIME-complete. Finally, we give a more efficient algorithm for computing the value of the delay in the case of words.

Unpublished Content Our notion of streamability, and the corresponding computational model, as presented in Chapter 3 have not been published yet. It is also the case for our streamable fragments of XPath, and the corresponding translation to dSTAs, presented in Chapter 6.

Author's Publications

[GCNT08] Olivier Gauwin, Anne-Cécile Caron, Joachim Niehren, and Sophie Tison. Complexity of Earliest Query Answering with Streaming Tree Automata. In *ACM SIGPLAN Workshop on Programming Language Techniques for XML (PLAN-X)*, January 2008. PLAN-X Workshop of ACM POPL. (Cited page 20)

[GNR08] Olivier Gauwin, Joachim Niehren, and Yves Roos. Streaming Tree Automata. *Information Processing Letters*, 109(1):13–17, December 2008. (Cited page 20)

[GNT09a] Olivier Gauwin, Joachim Niehren, and Sophie Tison. Bounded Delay and Concurrency for Earliest Query Answering. In *3rd International Conference on Language and Automata Theory and Applications*, volume 5457 of *Lecture Notes in Computer Science*, pages 350–361. Springer Verlag, 2009. (Cited page 20)

[GNT09b] Olivier Gauwin, Joachim Niehren, and Sophie Tison. Earliest Query Answering for Deterministic Nested Word Automata. In *17th International Symposium on Fundamentals of Computer Theory*, volume of 5699 of *Lecture Notes in Computer Science, pages 121–132. Springer Verlag, 2009. (Cited page 20)*

Chapter 2

Schemas and Query Languages

Contents

In this chapter, we introduce the basic notions used throughout this manuscript. The structures we study are unranked trees on a finite alphabet. We present this model, together with some standard logics and automata models. Schemas are another standard formalism for defining tree languages. Finally, queries over unranked trees are introduced using different objects: automata or XPath expressions. We survey known query answering algorithms for these query classes.

2.1 Unranked Trees and Logics

We start with the definition of unranked trees, and the standard framework that relates tree logics to tree automata [TW68, Don70, Tho97, CDG$^+$07], now com-

monly used in the context of XML [Nev02b, Nev02a, Lib06, Sch07b].

2.1.1 Trees and Binary Encodings

We define unranked trees as trees over an unranked alphabet. We then present two encodings into binary trees, used to lift results for ranked trees to unranked trees.

Alphabet

An *unranked alphabet* Σ is a finite set of symbols. A *ranked alphabet* is a pair (Σ, ar) where Σ is a finite set of symbols, and *ar* a function associating to each symbol its arity: $ar\colon \Sigma \to \mathbb{N}_0$. Here we write $\mathbb{N}_0$ for the set of non-negative integers, and $\mathbb{N}$ for natural numbers. For convenience the arity will be sometimes left implicit in the notations.

Unranked Trees

Let Σ be an unranked alphabet. The set of *unranked trees* over Σ, denoted $\mathcal{T}_\Sigma$, is the least set such that $a(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$ if $a \in \Sigma$, $k \in \mathbb{N}_0$ and for all $1 \leq i \leq k$, $t_i \in \mathcal{T}_\Sigma$. In particular we always exclude the empty tree from the set of trees.

An unranked tree *language* over Σ is a subset of $\mathcal{T}_\Sigma$. Unranked trees will be the default class of structures we will consider in this manuscript, so in the following a tree (resp. a tree language) will denote an unranked tree (resp. an unranked tree language). With this definition, trees are finite, ordered and labeled.

The set of *nodes* of a tree $t \in \mathcal{T}_\Sigma$ is the following prefix-closed language over natural numbers $\mathbb{N}$:

$$\mathit{nod}(a(t_1, \ldots, t_k)) = \{\epsilon\} \cup \{i \cdot \pi \mid \pi \in \mathit{nod}(t_i)\}$$

where $w \cdot w'$ is the concatenation of the words w and w'. The node ϵ always corresponds to the *root* of the tree. We inductively define the function $\mathit{lab}^t\colon \mathit{nod}(t) \to \Sigma$ that maps each node to its label. If $t = a(t_1, \ldots, a_k)$ then $\mathit{lab}^t(\epsilon) = a$, and $\mathit{lab}^t(i \cdot \pi) = \mathit{lab}^{t_i}(\pi)$.

The *depth* of a tree is the length of its longest branch:

$$\mathit{depth}(t) = \begin{cases} 1 & \text{if } t = a \text{ with } a \in \Sigma \\ 1 + \max_{1 \leq i \leq k} \mathit{depth}(t_i) & \text{if } t = a(t_1, \ldots, t_k) \text{ with } k \geq 1 \end{cases}$$

Hedges

A *hedge* over Σ is a sequence of trees $(t_1, \ldots, t_k)$ with $t_i \in \mathcal{T}_\Sigma$, for some $k \in \mathbb{N}_0$ and $1 \leq i \leq k$. The set of hedges over Σ is thus defined as:

$$\mathcal{H}_\Sigma = \{(t_1, \ldots, t_k) \mid k \in \mathbb{N}_0 \text{ and } t_i \in \mathcal{T}_\Sigma \text{ for all } 1 \leq i \leq k\}$$

The set of nodes of a hedge is defined from the set of nodes of its trees:

$$\mathit{nod}((t_1, \ldots, t_k)) = \bigcup_{1 \leq i \leq k} \{i \cdot \pi \mid \pi \in \mathit{nod}(t_i)\}$$

Note that the hedge (t_1) is different from the tree t_1, and has a different set of nodes. We will sometimes consider the empty hedge $()$.

Ranked Trees

In the following we always deal with unranked trees, but sometimes use automata on ranked trees together with a binary encoding, to define unranked tree languages.

Given a ranked alphabet (Σ, ar), we define the set of ranked trees over (Σ, ar) as the least set $\mathcal{T}^r_\Sigma$ containing $f(t_1, \ldots, t_k)$ for each symbol f of arity k and $t_1, \ldots, t_k \in \mathcal{T}^r_\Sigma$. Binary trees are a special case of ranked trees, where all symbols have arity 0 or 2. We write $\mathcal{T}^{\mathit{bin}}_\Sigma$ for the set of binary trees over a ranked alphabet (Σ, ar).

Binary Encodings

Binary encodings are used to encode unranked trees over Σ into binary trees. Two of them are commonly used: the *first-child next-sibling* encoding, and the *Curryfication*. For other encodings, see for instance [MSV03, FGK03].

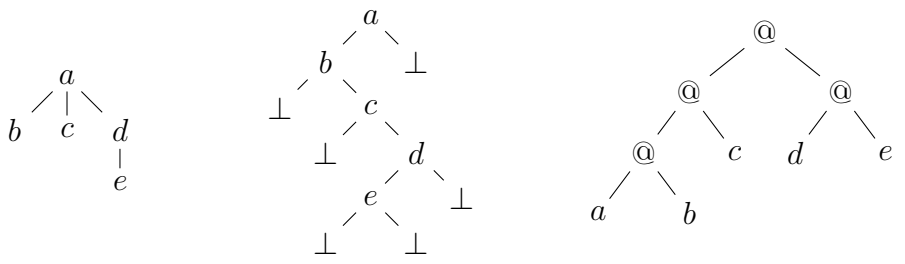
Rabin's first-child next-sibling encoding [Rab69, Koc03] is defined by $\mathit{fcns}\colon \mathcal{T}_\Sigma \to \mathcal{T}^{\mathit{bin}}_{\Sigma_\perp}$ where $\Sigma_\perp = \Sigma \uplus \{\perp\}$, all symbols from Σ having arity 2, and $\perp$ being the sole constant symbol. This is defined by the following rules, and illustrated in Figure 2.1(b). For convenience we first encode hedges into binary trees using $\mathit{fcns}_{\mathcal{H}}$:

$$\mathit{fcns}_{\mathcal{H}}(()) = \perp$$
$$\mathit{fcns}_{\mathcal{H}}((a(t'_1, \ldots, t'_m), t_2, \ldots, t_k)) = a(\ \mathit{fcns}_{\mathcal{H}}(t'_1, \ldots, t'_m)\ ,\ \mathit{fcns}_{\mathcal{H}}((t_2, \ldots, t_k))\ )$$

Then we simply use $\mathit{fcns}_{\mathcal{H}}$ on unary hedges: $\mathit{fcns}(t) = \mathit{fcns}_{\mathcal{H}}((t))$.

The second encoding of unranked trees corresponds to the Curryfication of terms, illustrated in Figure 2.1(c). This is defined through the function $\mathit{curry}\colon \mathcal{T}_\Sigma \to \mathcal{T}^{\mathit{bin}}_{\Sigma_@}$, where $\Sigma_@ = \Sigma \uplus \{@\}$ is the ranked alphabet in which all symbols from Σ are constant symbols, and $@$ is the only binary symbol.

$$\mathit{curry}(a(t_1, \ldots, t_k)) = \begin{cases} a & \text{if } k = 0 \\ @(\ \mathit{curry}(a(t_1, \ldots, t_{k-1}))\ ,\ \mathit{curry}(t_k)\ ) & \text{otherwise} \end{cases}$$



(a) A tree $t \in \mathcal{T}_\Sigma$. (b) *fcns*(t). (c) *curry*(t).

Figure 2.1: Binary encodings.

2.1.2 FO and MSO Logics

First-Order (FO) and Monadic Second-Order (MSO) logics are yardstick logics for expressing properties of structures. We start with the definition of relational structures, exhibit relational structures corresponding to unranked trees, and finally define the syntax and semantics of both logics.

Logics over unranked trees were recently surveyed by Libkin [Lib06] and Bojańczyk [Boj08]. In this manuscript we only address finite trees. More general results about finite models are available in the framework of finite model theory [EF99, Lib04].

Relational Structures

A *relational signature* Δ consists of a finite set of relation symbols $r \in \Delta$, each relation having a fixed arity $\mathit{ar}(r) \in \mathbb{N}_0$. A relational *structure* s over Δ consists of a non-empty finite set $\mathit{dom}(s)$ called the domain of s and relations $r^s \subseteq \mathit{dom}(s)^{\mathit{ar}(r)}$ interpreting all symbols $r \in \Delta$. We write $\mathcal{S}_\Delta$ for the set of structures over Δ. The size $|s|$ of a relational structure s is defined by: $|s| = |\mathit{dom}(s)| + |r^s|$.

Words as Relational Structures

We illustrate the definitions in the case of word structures. The signature, that we consider for words over a finite alphabet Σ, is $\Delta = \{\mathit{lab}_a \mid a \in \Sigma\} \cup \{\leq\}$. A non-empty word $w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*$ is the relational structure with domain $\mathit{dom}(w) = \{1, \ldots, k\}$ and the following relations:

- $\mathit{lab}_a^w = \{i \mid a_i = a,\ 1 \leq i \leq k\}$
- $\leq^w = \{(i, j) \mid 1 \leq i \leq j \leq k\}$

Trees as Relational Structures

An unranked tree $t \in \mathcal{T}_\Sigma$ can be also considered as a relational structure over the relational signature $\Delta = \{\mathit{lab}_a \mid a \in \Sigma\} \cup \{\mathit{fc}, \mathit{ns}\}$, where lab_a are monadic (i.e., unary) relations, while *fc* and *ns* are binary. The domain of t is exactly its set of nodes: $\mathit{dom}(t) = \mathit{nod}(t)$. The relations of the structure t are the following, where $a \in \Sigma$:

- $\mathit{lab}_a^t = \{\pi \mid \mathit{lab}^t(\pi) = a\}$
- $\mathit{fc}^t = \{(\pi, \pi{\cdot}1) \mid \pi{\cdot}1 \in \mathit{nod}(t)\}$
- $\mathit{ns}^t = \{(\pi{\cdot}i, \pi{\cdot}(i{+}1)) \mid 1 \leq i,\ \pi{\cdot}(i{+}1) \in \mathit{nod}(t)\}$

A tree t also defines the following relations, that we will sometimes use as base relations of some logics. *ch* is the standard *child* relation. ch^* (resp. ns^*) is the reflexive transitive closure of *ch* (resp. *ns*).

- $\mathit{ch}^t = \{(\pi, \pi{\cdot}i) \mid \pi{\cdot}i \in \mathit{nod}(t)\}$
- $(\mathit{ch}^*)^t = \{(\pi, \pi{\cdot}\pi') \mid \pi{\cdot}\pi' \in \mathit{nod}(t)\}$
- $(\mathit{ns}^*)^t = \{(\pi{\cdot}i, \pi{\cdot}j) \mid 1 \leq i \leq j,\ \pi{\cdot}j \in \mathit{nod}(t)\} \cup \{(\epsilon, \epsilon)\}$

Throughout the manuscript we use monadic predicates, selecting respectively the root node, the leaves, and the last children:

- $\mathit{root}^t = \{\epsilon\}$
- $\mathit{leaf}^t = \{\pi \in \mathit{nod}(t) \mid \nexists \pi'.\ (\pi, \pi') \in \mathit{ch}^t\}$
- $\mathit{lc}^t = \{\pi \in \mathit{nod}(t) \mid \nexists \pi'.\ (\pi, \pi') \in \mathit{ns}^t\}$

First-Order Logic

From a relational signature Δ and a countable set $\mathcal{V}$ of variables, the set $\mathrm{FO}[\Delta]$ of first-order formulas ϕ over Δ is defined by the following grammar:

$$\phi \ ::= \ r(x_1, \ldots, x_k) \mid \phi \wedge \phi \mid \neg\phi \mid \exists x.\ \phi \mid x = x'$$

where $r \in \Delta$ is a relation of arity k, and $x, x', x_1, \ldots, x_k \in \mathcal{V}$. Free variables of a formula ϕ are variables of $\mathcal{V}$ that appear in ϕ outside the scope of quantifiers $\exists$. Non-free variables are called bound variables in the following. A formula without free variables is called *closed*.

A formula $\phi \in \mathrm{FO}[\Delta]$ is interpreted over a relational structure s on the signature Δ using an assignment μ of the free variables of ϕ into $\mathit{dom}(s)$. The semantics

of FO[Δ]-formulas is defined through the satisfiability relation $s, \mu \models \phi$, as defined inductively below:

$s, \mu \models r(x_1, \ldots, x_k)$	iff	$(\mu(x_1), \ldots, \mu(x_k)) \in r^s$
$s, \mu \models \phi \wedge \phi'$	iff	$s, \mu \models \phi$ and $s, \mu \models \phi'$
$s, \mu \models \neg\phi$	iff	$s, \mu \not\models \phi$
$s, \mu \models \exists x.\ \phi$	iff	there exists $\pi \in \mathit{dom}(s)$ such that $s, \mu[x \leftarrow \pi] \models \phi$
$s, \mu \models x = x'$	iff	$\mu(x) = \mu(x')$

where $\mu[x \leftarrow \pi]$ is obtained from μ by assigning π to x.

Several signatures can be considered for the FO logic over unranked trees. The most commonly used is FO[$\mathit{ch}^*, \mathit{ns}^*$]. For convenience we always omit to mention the relations $(\mathit{lab}_a)_{a\in\Sigma}$, as they will always be part of the signature. This signature allows to define the relations *ch* and *ns*:

$$\begin{aligned} \mathit{ch}(x,y) &= \mathit{ch}^*(x,y) \wedge x \neq y \wedge \neg\exists z.\ z \neq x \wedge z \neq y \wedge \mathit{ch}^*(x,z) \wedge \mathit{ch}^*(z,y) \\ \mathit{ns}(x,y) &= \mathit{ns}^*(x,y) \wedge x \neq y \wedge \neg\exists z.\ z \neq x \wedge z \neq y \wedge \mathit{ns}^*(x,z) \wedge \mathit{ns}^*(z,y) \end{aligned}$$

On the contrary, the relations ch^* and ns^* are not definable in FO[*ch*, *ns*] [Lib04]. In the general case, FO does not allow to express the transitive closure of binary relations [Fag75, EF99].

The first-order logic is one of the key topics in logics and mathematics. For tree structures, numerous results have been established, even though some problems remain open. We outline the most relevant results in the following.

The *satisfiability* problem of a logic is the problem of deciding whether, given a formula ϕ in the logic, there exists a model for ϕ, i.e. a structure *s* and an assignment μ such that $s, \mu \models \phi$. While the satisfiability of FO formulas was proved undecidable for arbitrary [Chu36, Tur37] and finite structures [Tra50], it is decidable for trees (both ranked and unranked). This also holds for the Monadic Second-Order logic, an extension of FO that we present below.

The *model-checking* problem is the decision problem that takes as input a structure *s*, an assignment μ and a formula ϕ, and outputs the truth value of $s, \mu \models \phi$. For FO on finite structures, the model-checking is PSPACE-complete, even on trees [Sto74, Var82].

Algebraic characterizations of FO-definable tree languages (for instance by means of automata) are more complex than for the MSO logic. Some work on this topic can be found in the manuscript of Bojańczyk [Boj04]. In [BS05], Benedikt and Segoufin study the FO-definability problem, i.e. the problem of deciding whether a tree language can be defined using an FO formula. They present such a procedure for FO[*ch*, *ns*] over ranked trees and unordered unranked trees. The question is still open for ordered unranked trees, the class of structures that we consider in this manuscript.

Monadic Second-Order Logic

The Monadic Second-Order logic (MSO) extends the First-Order logic with quantification over second-order variables, i.e. unary predicates, that are usually interpreted as sets. We extend $\mathcal{V}$ with second-order variables, ranged over by X. MSO[Δ] is the set of MSO formulas over the signature Δ, as defined by the grammar:

$$\phi \ ::= \ r(x_1, \ldots, x_k) \ \mid \ \phi \wedge \phi \ \mid \ \neg\phi \ \mid \ \exists x.\ \phi \ \mid \ \exists X.\ \phi \ \mid \ x \in X$$

where $r \in \Delta$ has arity k, and $x, x_1, \ldots, x_k, X \in \mathcal{V}$.

The semantics of FO formulas can be easily extended to MSO. It is now defined on a structure s under an assignment μ, that maps each free first-order variable to an element of $\mathit{dom}(s)$ and each free second-order variable to a subset of $\mathit{dom}(s)$. Then the satisfiability relation is extended in the following way:

$$\begin{array}{lll} s, \mu \models \exists X.\ \phi & \text{iff} & \text{there exists } D \subseteq \mathit{dom}(s) \text{ such that } s, \mu[X \leftarrow D] \models \phi \\ s, \mu \models x \in X & \text{iff} & \mu(x) \in \mu(X) \end{array}$$

For unranked tree structures, the usual signature used for expressing MSO formulas is $\Delta = \{\mathit{fc}, \mathit{ns}, (\mathit{lab}_a)_{a\in\Sigma}\}$, and we denote the corresponding logic by MSO[fc, ns]. Unlike FO logic, MSO can express the transitive closure of binary relations. For instance the following formula ϕ is the transitive closure of the relation defined by φ:

$$\phi(y_1, y_2) = \forall X.\ (y_1{\in}X \wedge \forall(x_1, x_2).\ (x_1{\in}X \wedge \varphi(x_1, x_2) \ \Rightarrow \ x_2{\in}X)) \ \Rightarrow \ y_2{\in}X$$

Hence we can define ns^* from *ns*, then *ch* by composing *fc* and ns^*, and finally ch^* from *ch*. A tree language L is said *MSO-definable* if there exists an MSO[fc, ns]-formula ϕ without free variable such that

$$L = \{t \in \mathcal{T}_\Sigma \ \mid \ t \models \phi\}$$

On binary trees, MSO is sometimes called the *weak second order logic with two successors* (WS2S): the two successor relations are first-child and second-child, and *weak* means that the second-order variables are interpreted as *finite* sets. WSkS is the generalization to k successors.

MSO enjoys clean algebraic characterizations, as opposed to known FO characterizations [Boj04]. The first link with automata was made by Büchi on strings [Büc60]. In the following, we introduce tree automata and recall the equivalence between tree automata and MSO on trees, as established by Doner [Don70], and Thatcher and Wright [TW68]. This translation comes at a certain cost, having the following consequences on satisfiability and model-checking problems.

Satisfiability of MSO[*fc*, *ns*] formulas is known to be non-elementary [SM73, Mey73, Sto74]: for every algorithm solving this problem, its complexity cannot be bounded by a tower of exponential of fixed height [Grz53, FG02]. A way to test the satisfiability is by translation of formulas into tree automata, which is a non-elementary process. Then it suffices to test the emptiness of tree automata, which can be done in PTIME, as shown in Section 2.1.3.

The model-checking of MSO[*fc*, *ns*] formulas on finite trees is a PSPACE-complete problem, as for FO formulas [Sto74, Var82]. When the formula is fixed, the problem becomes linear, as we can translate the formula into an automaton in constant time (disregarding thus the non-elementary blowup), and then check that the tree is accepted by the automaton in linear time.

2.1.3 Tree Automata

Unranked trees can be converted into ranked ones using encodings, as shown in Section 2.1.1. We introduce tree automata for binary trees, and present the language of unranked trees they define, when associated with a binary encoding.

Tree automata were introduced by Doner [Don65, Don70] and Thatcher and Wright [TW65, TW68], to prove the decidability of the weak second order theory of multiple successors (WSkS). They regained interest in the context of XML, as shown in the surveys by Neven [Nev02b, Nev02a] and Schwentick [Sch07b].

Bottom-Up Tree Automata

Let $\Sigma_r = \Sigma_0 \uplus \Sigma_2$ be a ranked alphabet, where arity of symbols in Σ_0 (resp. Σ_2) is 0 (resp. 2). A (bottom-up) *tree automaton* (TA) for binary trees in $\mathcal{T}^{bin}_{\Sigma_r}$ is a tuple $A = (\mathit{stat}, \mathit{fin}, \mathit{rul})$ consisting of finite sets $\mathit{fin} \subseteq \mathit{stat}$ and a set $\mathit{rul} \subseteq \mathit{stat} \times \Sigma_0 \cup \mathit{stat}^3 \times \Sigma_2$, that we denote as

$$f(q_1, q_2) \to q \qquad \text{and} \qquad c \to q$$

where $q_1, q_2, q \in \mathit{stat}$, $f \in \Sigma_2$ and $c \in \Sigma_0$. A run of A on $t \in \mathcal{T}^{bin}_{\Sigma_r}$ is a function r: $\mathit{nod}(t) \to \mathit{stat}$ such that $f(r(\pi\cdot 1), r(\pi\cdot 2)) \to r(\pi)$ belongs to *rul* for all nodes π of t with $\mathit{lab}^t(\pi) = f \in \Sigma_2$, and $c \to r(\pi)$ in *rul* for all nodes π of t with $\mathit{lab}^t(\pi) = c \in \Sigma_0$. A run is *successful* if $r(\epsilon) \in \mathit{fin}$. The language $L^{bin}(A)$ is the set of all binary trees over Σ_r that permit a successful run by A. Doner [Don70] and Thatcher and Wright [TW68] proved that a ranked tree language is recognizable by a TA iff it can be defined in the WS2S logic. WS2S corresponds to MSO with a monadic predicate for label tests and two binary predicates, one for the left child, and one for the right one.

A (bottom-up) deterministic TA (dTA) is a TA that does not have two rules with the same left-hand side. TA are determinizable, i.e. every TA A has an equivalent dTA A'. The determinization procedure has an EXPTIME lower bound.

The size of a TA A is its number of states plus its number of rules: $|A| = |\mathit{stat}_A| + |\mathit{rul}_A|$. We sometimes provide complexity results in terms of number of states $|\mathit{stat}_A|$, number of rules $|\mathit{rul}_A|$, or size of the alphabet $|\Sigma|$, whenever this precision is relevant.

When associated with a binary encoding, these automata define languages of unranked trees:

$$L_{\mathit{enc}}(A) = \{t \in \mathcal{T}_\Sigma \mid f(t) \in L^{\mathit{bin}}(A)\}$$

with $\mathit{enc} \in \{\mathit{fcns}, \mathit{curry}\}$. Stepwise tree automata [CNT04] are exactly TAs used with the *curry* encoding.

A language L of binary trees (resp. unranked trees) is *regular* if there is an automaton A for binary trees such that $L^{\mathit{bin}}(A) = L$ (resp. $L_{\mathit{fcns}}(A) = L$). Here we choose *fcns* as binary encoding, but we will see in Chapter 4 that choosing *curry* defines the same class.

Top-Down Tree Automata

Numerous other automata notions were defined. In the ranked case, we mention *top-down tree automata* (↓TA) [CDG$^+$07], as we will use them later on to capture some schema languages. ↓TAs are similar to TAs, but evaluates the tree by starting at the root and ending in leaves.

A top-down tree automaton (↓TA) for binary trees in $\mathcal{T}^{\mathit{bin}}_{\Sigma_r}$ is syntactically equivalent to a bottom-up TA. However, the corresponding notion of runs differ, and for clarity we choose to represent the rules as

$$q, f \to (q_1, q_2) \qquad \text{and} \qquad q \to c$$

for binary symbols $f \in \Sigma_2$ and symbols $c \in \Sigma_0$. A run of a ↓TA A on a tree $t \in \mathcal{T}^{\mathit{bin}}_{\Sigma_r}$ is also a function r: $\mathit{nod}(t) \to \mathit{stat}$, but evaluated from root to leaves: For all nodes π of t, $r(\pi), f \to (r(\pi \cdot 1), r(\pi \cdot 2)) \in \mathit{rul}$ if $\mathit{lab}^t(\pi) = f \in \Sigma_2$, and $r(\pi) \to c \in \mathit{rul}$ if $\mathit{lab}^t(\pi) = c \in \Sigma_0$. A run is accepting if $r(\epsilon) \in \mathit{init}$. Hence $L^{\mathit{bin}}(A)$ is the set of trees for which a run of A exists. As usual, ↓TA can be used together with a binary encoding to define a language of unranked trees. Deterministic ↓TAs (d↓TAs) are ↓TAs having at most one right hand side per left hand side in its rules, and a unique initial state. d↓TAs are known to be strictly less expressive than ↓TAs, while ↓TAs are as expressive as TAs [CDG$^+$07].

Alternatives

Tree Walking Automata (TWAs) [AU71] are automata for ranked trees, that do not operate in parallel, nor use a stack. They run through the tree from one node to another, according to the direction indicated by the rule. TWAs are strictly less expressive than TAs [BC05]. They are even less expressive than FO extended with a transitive closure operator [EH07]. However their nested variant was used to prove that this extension of FO is strictly less expressive than MSO [BSSS06, tCS08]. TWAs cannot be determinized [BC04]. Some extensions of TWAs with pebbles define a hierarchy of automata classes, with different expressiveness [EH99, EHB99, BSSS06].

For unranked trees, many models were proposed too, as surveyed in [CDG$^+$07, Sch07b] for instance. One of the first model designed for processing XML documents are hedge automata [BKWM01]. Hedge automata operate bottom-up, and use a regular language as acceptor for the language of children of a node.

Chapter 4 of this manuscript introduces Streaming Tree Automata, a model where trees are evaluated using a pre-order traversal of their structure. In that chapter we exhibit the links with other models that use this evaluation order, on structures that include unranked trees: Visibly Pushdown Automata [AM04], Nested Word Automata [Alu07] and Pushdown Forest Automata [NS98].

Expressiveness and Closure Properties

Doner [Don70] and Thatcher and Wright [TW68] proved that the class of regular ranked tree languages is exactly the class of MSO-definable ranked tree languages. It is folklore that this equivalence also holds in the unranked case [CDG$^+$07].

Proposition 1. *A language $L \subseteq \mathcal{T}_\Sigma$ is MSO-definable iff it is regular.*

Hence closure properties of MSO-definable languages also apply to regular languages [CDG$^+$07].

Proposition 2. *Regular languages are closed under complement, union, and intersection. The corresponding operations on TAs can be done in* PTIME*, except the complementation of non-deterministic automata. They all preserve determinism except the projection.*

We recall the complexity of some decision problems for tree automata. These results hold for both ranked and unranked tree automata.

problem	input	output	complexity for TAs	complexity for dTAs
emptiness	A	$L(A) = \emptyset$?	$O(\|A\|)$	$O(\|A\|)$
universality	A	$L(A) = \mathcal{T}_\Sigma$?	EXPTIME-complete	PTIME
inclusion	A, A'	$L(A) \subseteq L(A')$?	EXPTIME-complete	$O(\|A\| \cdot \|A'\|)$

Note that for the inclusion problem, only A' needs to be deterministic. The usual technique is to test whether $L(A \cap \overline{A'}) = \emptyset$, where $\overline{A'}$ is the complement of A'. This complementation might imply higher complexity, as it requires completion. However this completion can be avoided [CGLN09].

2.2 Schemas

Schema languages are used to define sets of *valid* trees. In the context of XML, schemas are used to specify the possible structures of trees that represent some set of documents. Schema languages are often based on tree grammars [MLM01], but here we consider them from the perspective of tree automata. In this manuscript we study some schema languages, that will be useful in the context of a streaming evaluation on XML documents. We restrict ourselves to Document Type Definitions (DTDs) and their extended version. Other standard schema languages are, for instance, XML Schema [FW04, MLM01, Chi00], Relax NG [CM01] and Schematron [Jel06]. Note that both XML Schemas and Relax NG can be modeled by Extended DTDs. For a more complete description and study of schema languages, we refer the reader to [MLM01, MNSB06b, Sch07b, CDG$^+$07].

2.2.1 Document Type Definition

The Document Type Definition (DTDs) is a W3C recommendation [BPSM$^+$08], and the most commonly used formalism for defining schemas over XML documents. A DTD is an extended context-free grammar, i.e. a context-free grammar where right-hand sides are regular expressions. Figure 2.2 contains an example of DTD for documents describing discotheques. The XML document in Figure 2.3 is valid w.r.t. to this DTD. Real DTDs permit the use of the #PCDATA symbol, indicating that some textual data is expected. Here we replace it by ϵ as we never take data values into account in this manuscript.

Formally, a DTD D over the alphabet Σ is a pair $D = (\mathit{init}, \mathit{rul})$, where $\mathit{init} \in \Sigma$ is a start symbol, and *rul* a function mapping a regular expression $e = \mathit{rul}(a)$ for every symbols $a \in \Sigma$. For convenience we often write *rul* as a set of mappings $a \to e$. Regular expressions respect the following grammar:

$$e ::= a \mid e{\cdot}e \mid e + e \mid e^* \mid \epsilon$$

where $a \in \Sigma$ and ϵ is the empty word. We write $L(e) \subseteq \Sigma^*$ for the word language defined by the regular expression e. Then for each letter $a \in \Sigma$, the DTD inductively defines the following set of unranked trees:

$$L_a(D) = \{a(t_1, \ldots, t_k) \mid a_1 \ldots a_k \in L(\mathit{rul}(a)),\ t_i \in L_{a_i}(D) \text{ for } 1 \leq i \leq k\}$$

$$
\begin{array}{lcl}
\text{albums} & \rightarrow & (\text{cd} + \text{online})^* \\
\text{cd} & \rightarrow & \text{title} \cdot \text{author} \cdot \text{tracklist} \\
\text{online} & \rightarrow & \text{title} \cdot \text{author} \cdot \text{tracklist} \cdot \text{url} \\
\text{tracklist} & \rightarrow & \text{track} \cdot \text{track}^* \\
\text{title} & \rightarrow & \#\text{PCDATA} \\
\text{author} & \rightarrow & \#\text{PCDATA} \\
\text{track} & \rightarrow & \#\text{PCDATA} \\
\text{url} & \rightarrow & \#\text{PCDATA}
\end{array}
$$

Figure 2.2: A DTD describing discotheques.

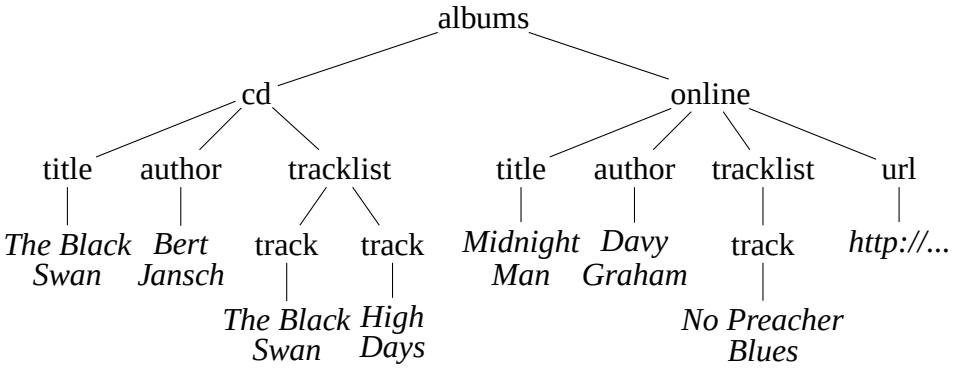


Figure 2.3: A valid tree describing a discotheque.

The language of valid trees defined by the DTD $D = (\mathit{init}, \mathit{rul})$ is the language associated with its start symbol, i.e. $L_{\mathit{init}}(D)$.

Expressiveness

DTDs are strictly less expressive than regular languages. They exactly correspond to *local tree languages* [MLM01]: for every pair of valid trees t and t', if t (resp. t') has a node π (resp. π') labeled by $a \in \Sigma$, then replacing in t the subtree rooted at π by the subtree of t' rooted at π' leads to a new valid tree. In other terms, DTDs do not take the context into account, but only the local label [PV00]. Hence, DTDs can be translated in PTIME to $\downarrow$TAs recognizing the *fcns* encoding of valid trees. A lot of algorithms were proposed for processing efficiently DTDs with regards to the usual problems related to tree languages: membership (here, named validation) and typing [BKW98, SV02], inclusion, equivalence [MNS04].

Beside this formalization, the W3C recommendation [BPSM$^+$08] indicates that the regular expressions have to be *one-unambiguous*. This means that when parsing the word from left to right, there must be at any time point at most one possible matching in the regular expression. In other terms, the Glushkov automaton [Glu61] obtained from the regular expression must be deterministic. We call a DTD *deterministic*, if all its corresponding regular expressions are one-unambiguous.

2.2.2 Extended Document Type Definition

Extended DTDs (EDTDs for short, and sometimes called *specialized* DTDs in the literature) were proposed by Papakonstantinou and Vianu [PV00], by allowing each label to have several types. Each type is associated with one label. The regular expressions of an EDTD are not based on labels, but on types. This way, EDTDs capture all regular languages.

For instance, consider the discotheque example previously introduced. Suppose that we want to use a url for authors instead of some #PCDATA, but only for online albums. This would be impossible using a DTD, as this is a non-local property. With EDTDs, we can introduce two types of authors, and thus solve the

problem:

$$\begin{array}{rl}
\text{albums} \to (\text{cd} \mid \text{online})^* & \textit{type}(\text{albums}) = \text{albums} \\
\text{cd} \to \text{title}\cdot\text{cdAuthor}\cdot\text{tracklist} & \textit{type}(\text{cd}) = \text{cd} \\
\text{online} \to \text{title}\cdot\text{onlineAuthor}\cdot\text{tracklist}\cdot\text{url} & \textit{type}(\text{online}) = \text{online} \\
\text{tracklist} \to \text{track}\cdot\text{track}^* & \textit{type}(\text{tracklist}) = \text{tracklist} \\
\text{title} \to \#\text{PCDATA} & \textit{type}(\text{title}) = \text{title} \\
\text{cdAuthor} \to \#\text{PCDATA} & \textit{type}(\text{cdAuthor}) = \text{author} \\
\text{onlineAuthor} \to \text{url} & \textit{type}(\text{onlineAuthor}) = \text{author} \\
\text{track} \to \#\text{PCDATA} & \textit{type}(\text{track}) = \text{track} \\
\text{url} \to \#\text{PCDATA} & \textit{type}(\text{url}) = \text{url}
\end{array}$$

More formally, an EDTD D over Σ is a tuple $(\textit{init}, \textit{rul}, \mathfrak{T}, \textit{type})$ where $\mathfrak{T}$ is the set of types, $\textit{init} \in \mathfrak{T}$, *rul* maps each type of $\mathfrak{T}$ to a regular expression of types, and *type* maps each type to a symbol of Σ. With each type $\vartheta \in \mathfrak{T}$ we can associate the language:

$$L_\vartheta(D) = \left\{ a(t_1, \ldots, t_k) \mid \begin{array}{l} a = \textit{type}(\vartheta), \\ \vartheta_1 \ldots \vartheta_k \in L(\textit{rul}(\vartheta)),\ t_i \in L_{\vartheta_i}(D) \text{ for } 1 \leq i \leq k \end{array} \right\}$$

The language recognized by D is $L_{\textit{init}}(D)$. In terms of expressiveness, EDTDs exactly capture the set of regular unranked tree languages [PV00].

Introducing types leads to the problem of typing each label of a document. Two types are said *competing* if both are mapped to the same label (for instance, *cdAuthor* and *onlineAuthor* in our example). Computing types increases the cost of parsing and processing, when compared to DTDs. This is why some restrictions on EDTDs have been proposed.

Single-type EDTDs

The first restriction on EDTDs is to require that no regular expression can contain two competing types. This corresponds to *single-type* EDTDs, and also to XML Schema according to [MLM01] (see also [MNSB06b]). Single-type EDTDs is also the class of languages for which the ancestor string (the concatenation of labels of the current branch) determines the type: if two valid trees have the same ancestor strings until nodes π and π', then swapping the corresponding subtrees leads also to valid trees [MNS05].

In our discography example, the EDTD extension is single-type, as the only competing types are *cdAuthor* and *onlineAuthor*, and they never appear in the same rule.

Restrained Competition EDTDs

We introduce the second restriction, namely the *restrained competition*. This one is similar to the determinism of DTDs, but at the level of types: An EDTD is *restrained competition* if there does not exist two different competing types ϑ_1 and ϑ_2 and words $u, v_1, v_2 \in \mathfrak{T}^*$ such that $\{u \cdot \vartheta_1 \cdot v_1,\ u \cdot \vartheta_2 \cdot v_2\} \in L(e)$ for some regular expression e in *rul*. Martens et al. [MNS05] prove that deciding whether an EDTD is restrained-competition is in (a subclass of) PTIME. An EDTD is *deterministic* if all its regular expressions are one-unambiguous. Clearly, every single type EDTD is also restrained competition, and every restrained competition EDTD is deterministic.

Restrained competition EDTDs are strictly more expressive than deterministic DTDs, but strictly less than regular languages [MLM01]. In fact, we get the same characterization as for single-type EDTDs, except that we replace the string of ancestors by the string of ancestors of the leftmost sibling of the node, plus its left siblings. Hence deterministic restrained competition EDTDs can be translated in linear time to d↓TAs on the first-child next-sibling encoding of trees (see for instance Lemma 33 of [CGLN09]). Deterministic restrained competition EDTDs can be efficiently used to type documents in streaming order. In Chapter 4, we present a translation of restrained-competition EDTDs to automata that evaluate documents in a streaming fashion.

2.3 Queries

In the context of databases, queries are used to select data to be processed later on. In this manuscript, we focus on queries that only take the structure of the database into account, not the data values.

We define n-ary queries over relational structures, as functions selecting n-tuples of elements of the domain. The special cases of queries over words and trees are introduced. Logics and automata, as presented previously, are then used for defining n-ary queries. Finally, the W3C standards XPath 1.0 and XPath 2.0 are introduced, and their navigational cores are formalized. We also mention other formalisms for querying in trees, and expose the state of the art for queries evaluation.

2.3.1 Queries over Relational Structures

We first introduce queries over relational structures. In the context of XML, schemas are used to define the set of valid trees. In this manuscript, we study the evaluation of queries that only select tuples of nodes in valid trees of some

given schema. To generalize this idea, queries are always given with an associated schema, that we name the *domain* of the query. This has to be distinguished with the set of trees on which the query selects some nodes, and thus the schema given by a separate object.

Definition

Let Δ be a relational signature and $n \in \mathbb{N}_0$. A *schema over* Δ is a subset $S \subseteq \mathcal{S}_\Delta$. An *$n$-ary query with schema* S is a function Q with domain $\mathit{dom}(Q) = S$, which maps all structures $s \in S$ to a set of tuples of elements, and only selects on valid structures:

$$Q(s) \subseteq \mathit{dom}(s)^n \quad \text{and} \quad Q(s) \neq \emptyset \Rightarrow s \in \mathit{dom}(Q)$$

A *Boolean query* Q is a query of arity 0, where the empty tuple $()$ is selected for some trees. A *monadic query* is a query of arity 1. We sometimes use queries *without schema*, meaning that we consider queries with the universal schema $S = \mathcal{S}_\Delta$.

A *query language* (also called *query class* in this manuscript) $\mathcal{Q}$ of arity n over Δ consists of a set $\mathcal{Q}$, whose expressions $e \in \mathcal{Q}$ have a size $|e| \in \mathbb{N}$ and a query Q_e of arity n, so that $Q_e(s) \subseteq \mathit{dom}(s)^n$ for all $s \in \mathcal{S}_\Delta$. Note that the expression e defines both the schema $\mathit{dom}(Q_e) \subseteq \mathcal{S}_\Delta$ and the object for selecting nodes $Q_e(s) \subseteq \mathit{dom}(s)^n$. Hence expressions are usually a pair of objects. In this manuscript we will study query classes for which expressions will be either XPath expressions or tree automata for selecting nodes, with automata for the schema languages.

The query evaluation problem takes as inputs an expression e and a structure s, and outputs $Q_e(s)$. It is parameterized by a query class. The complexity of this problem when the query and structure are both variable, is called *combined complexity*. When the size of the expression is fixed, we name it *data complexity*.

Below, we will define queries in words, where the schema is a class of relational structures of words in $\mathit{dom}(Q) \subseteq \Sigma^*$, and queries in unranked trees where the schema is a class of relational structures of unranked trees $\mathit{dom}(Q) \subseteq \mathcal{T}_\Sigma$. The domains can be defined by automata or XML schemas.

FO and MSO-definable Queries

Queries can be easily defined from FO and MSO formulas, by using their free variables. This can be done modulo an ordering on these free variables, and by requiring that MSO formulas only have first-order free variables.

Let $\phi, \phi' \in \mathrm{FO}[\Delta]$ (resp. $\phi, \phi' \in \mathrm{MSO}[\Delta]$) where ϕ' is closed, and let $x_1, \ldots, x_n$ be the free variables of ϕ, all of them being first-order. Then we define

the n-ary query $Q_{\phi(x_1,\ldots,x_n),\phi'}$ by:

$$Q_{\phi(x_1,\ldots,x_n),\phi'}(\mathcal{S}) = \{(\pi_1,\ldots,\pi_n) \mid \mathcal{S}, [x_1 \leftarrow \pi_1,\ldots,x_n \leftarrow \pi_n] \models \phi\}$$

for all $\mathcal{S} \in \mathcal{S}_\Delta$ such that $\mathcal{S} \models \phi'$, and $\mathit{dom}(Q_{\phi(x_1,\ldots,x_n),\phi'}) = \{\mathcal{S} \mid \mathcal{S} \models \phi'\}$. Similarly, we define $Q_{\phi(x_1,\ldots,x_n)}$ for the case without schema, by lifting the condition $\mathcal{S} \models \phi'$ and $\mathit{dom}(Q_{\phi(x_1,\ldots,x_n),\phi'}) = \mathcal{S}_\Delta$.

We say that an n-ary query Q is *FO-definable* (resp. *MSO-definable*) over Δ-structures if there exist FO[Δ] formulas (resp. MSO[Δ] formulas) ϕ with free variables $x_1,\ldots,x_n$ and ϕ' (a closed formula) such that $Q = Q_{\phi(x_1,\ldots,x_n),\phi'}$. Hence FO[$\Delta$] and MSO[$\Delta$] are two query classes, whose expressions are formulas with ordered free variables for the selecting part, with closed formulas for the schema part.

Canonical Language

We can equivalently define a query as a set of annotated structures. This will be used to define queries by structures acceptors, like automata. Boolean queries Q with $\mathit{dom}(Q) = \mathcal{S}_\Delta$ can be identified with structures $L_Q = \{\mathcal{S} \mid () \in Q(\mathcal{S})\}$. But how can we define languages of structures for n-ary queries?

We fix an ordered set of distinct variables $\mathcal{V}_n = \{x_1,\ldots,x_n\}$ and define extended relation signatures $\Delta_n = \Delta \cup \mathcal{V}_n$ such that every variable becomes a unary relation symbol. For every structure $\mathcal{S} \in \mathcal{S}_\Delta$ and tuple $\tau = (\pi_1,\ldots,\pi_n) \in \mathit{dom}(\mathcal{S})^n$ we define an *annotated structure* $\mathcal{S} * \tau \in \mathcal{S}_{\Delta_n}$ as follows:

$$\begin{array}{ll} \mathit{dom}(\mathcal{S} * \tau) = \mathit{dom}(\mathcal{S}) & \\ r^{\mathcal{S}*\tau} = r^{\mathcal{S}} & \text{for all } r \in \Delta \\ x_i^{\mathcal{S}*\tau} = \{\pi_i\} & \text{for all } 1 \leq i \leq n \end{array}$$

We call a structure $\tilde{s} \in \mathcal{S}_{\Delta_n}$ *canonical* if $x^{\tilde{s}}$ is a singleton for all $x \in \mathcal{V}_n$. Clearly, all annotated structures $\mathcal{S} * \tau$ are canonical. Conversely, every canonical structure $\tilde{s}$ is equal to some annotated structure $\mathcal{S} * \tau$. We therefore define the canonical language L_Q of an n-ary query Q as the following set of annotated structures:

$$L_Q = \{\mathcal{S} * \tau \mid \tau \in Q(\mathcal{S})\}$$

The canonical language of a Boolean query indeed coincides with the schema $L_Q = \{\mathcal{S} \mid () \in Q(\mathcal{S})\}$. Note however, that the domain of a query is only partially specified by the canonical language. In particular there may exist valid structures $\mathcal{S} \in \mathit{dom}(Q)$ on which nothing is selected, i.e., $Q(\mathcal{S}) = \emptyset$, so we cannot identify $\mathit{dom}(Q)$ with the structures on which something is selected. In order to fix this problem, we identify a Q with the pair $(L_Q, \mathit{dom}(Q))$ of its canonical language and its domain.

Logical Operations on Queries

We define logical operations for n-ary queries Q, Q' with the same schema S: conjunction $Q \wedge Q'$, disjunction $Q \vee Q'$, negation $\neg Q$, existential quantification $\exists x_i.\ Q$ and cylindrification $c_i Q$ for all $1 \leq i \leq n$. All these queries have the same domain S and satisfy for all structures $\mathcal{s} \in S$:

conjunction	$Q \wedge Q'(\mathcal{s}) = Q(\mathcal{s}) \cap Q'(\mathcal{s})$
negation	$\neg Q(\mathcal{s}) = \begin{cases} \mathit{dom}(\mathcal{s})^n - Q(\mathcal{s}) & \text{if } \mathcal{s} \in S \\ \emptyset & \text{otherwise} \end{cases}$
quantification	$\exists x_i.\ Q(\mathcal{s}) = \{(\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n) \mid \exists \pi_i.\ (\pi_1, \ldots, \pi_n) \in Q(\mathcal{s})\}$
cylindrification	$c_i Q(\mathcal{s}) = \{(\pi_1, \ldots, \pi_i, \pi, \pi_{i+1}, \ldots, \pi_n) \mid (\pi_1, \ldots, \pi_n) \in Q(\mathcal{s})\}$

Note that $\exists x_i.\ Q$ is a query of arity $n-1$ and $c_i Q$ arity $n+1$, while all others have arity n.

We next relate logical operations on queries to set operations on canonical languages. This correspondence is the reason why this annotation method is said *canonical*. We define for all $r \in \Delta_n$ a projection operator $\Pi_r \colon \mathcal{S}_\Delta \to \mathcal{S}_{\Delta_n - \{r\}}$ which removes symbol r from the relational structures. We get the following equalities:

intersection	$L_{Q \wedge Q'} = L_Q \cap L_{Q'}$
complement	$L_{\neg Q} = \{\mathcal{s} * \tau \mid \mathcal{s} \in \mathit{dom}(Q),\ \tau \in \mathit{dom}(\mathcal{s})^n\} - L_Q$
projection	$L_{\exists x.\ Q} = \{\Pi_x(s * \tau) \mid s * \tau \in L_Q\}$
cylindrification	$L_{c_i Q} = \bigcup_{s \in L_Q} \Pi_{x_i}^{-1}(s)$

Queries over Words

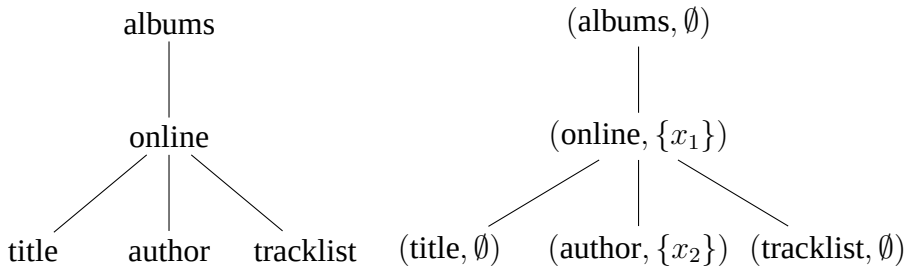
An n-ary query Q in words has some schema $\mathit{dom}(Q) \subseteq \Sigma^*$ and selects n-tuples of positions in words in $\mathit{dom}(Q)$. Suppose that we fix $\mathit{dom}(Q) = \Sigma^*$. We can then define a monadic query by the following FO-formula with a single free variable x_1:

$$\phi(x_1) = \exists x_2.\ (x_1 \leq x_2 \wedge \mathit{lab}_a(x_2))$$

For every word w in the schema, the query $Q_{\phi(x_1)}$ defined by this formula selects all positions before some a-labeled positions.

Given a word $w = a_1 \cdot \ldots \cdot a_m \in \Sigma^*$ and a tuple $\tau = (\pi_1, \ldots, \pi_n) \in \mathit{dom}(w)^n$, we can identify the annotated structure $w * \tau$ with the following annotated word over $\Sigma \times 2^{\mathcal{V}_n}$:

$$(a_1, \{x_i \mid \pi_i = 1\}) \cdot \ldots \cdot (a_m, \{x_i \mid \pi_i = m\})$$



(a) A tree $t \in \mathcal{T}_\Sigma$. (b) The annotated tree $t * (1, 1 \cdot 2)$.

Figure 2.4: Example of tree annotation.

For instance, we identify the annotated structure $b \cdot a \cdot c * (2)$ with the word $(b, \emptyset) \cdot (a, \{x_1\}), (c, \emptyset)$. Hence the canonical language of an n-ary query Q in words over Δ thus can be identified with a language L_Q of annotated words with alphabet $\Sigma \times 2^{\mathcal{V}_n}$.

Queries over Unranked Trees

Queries Q in unranked trees of $\mathcal{T}_\Sigma$ are queries with some domain $\mathit{dom}(Q) \subseteq \mathcal{T}_\Sigma$. They select tuples of nodes $Q(t) \subseteq \mathit{nod}(t)^n$ for all trees $t \in \mathit{dom}(Q)$. For instance, considering the schema in Figure 2.2 describing discotheques, we can define a query that selects all pairs of nodes (π, π') where π' is a descendant of π labeled by *author* using the following FO-formula with free variables x_1 and x_2:

$$\phi(x_1, x_2) = (\mathit{ch}^*(x_1, x_2) \wedge \mathit{lab}_{\mathit{author}}(x_2))$$

By analogy with the case of words, the canonical language of an n-ary query Q in unranked trees over Σ can be identified with a language of unranked trees over the alphabet $\Sigma \times 2^{\mathcal{V}_n}$ where $\mathcal{V}_n = \{x_1, \ldots, x_n\}$. We illustrate this in Figure 2.4.

2.3.2 Queries by Automata

Let $n \in \mathbb{N}_0$ be some arity. If a notion of automaton exists for a class of annotated relational structures $\mathcal{S}_{\Delta_n}$, then we can use automata for defining queries over structures of $\mathcal{S}_\Delta$ by means of the canonical languages.

If A is a tree automaton over the alphabet $\Sigma \times 2^{\mathcal{V}_n}$ recognizing only canonical structures and B a word (resp. tree) automaton over Σ, then we define the query $Q_{A,B}$ by:

$$L_{Q_{A,B}} = L(A) \qquad \text{and} \qquad \mathit{dom}(Q_{A,B}) = L(B)$$

Note that in the definition of queries, we required that queries only select on valid trees. This means that we only consider automata A, B such that $\Pi_\Sigma(L(A)) \subseteq L(B)$, where Π_Σ is the projection along the Σ component.

We call a query Q over unranked trees *regular* if there exist two tree automata A and B such that $Q = Q_{A,B}$. From these definitions, we can extend the correspondence between regular and MSO-definable tree languages of Proposition 1 to queries. It suffices to use Proposition 1 on the canonical language of Q.

Proposition 3. *A query over unranked trees is regular iff it is MSO-definable.*

More complete results about the links between logics and automata for trees are presented in [CDG$^+$07, Nev02b, Nev02a] and for more general structures in [Tho97].

Related Work on Queries by Automata

Different approaches were proposed to use automata to define queries on trees.

An alternative way of using automata for defining queries is to use the annotations of trees by runs of the automaton, where some tuples of states permit to define tuples of selected nodes. This can also be seen as putting the variables $\mathcal{V}_n$ in the states instead of the alphabet. Let *Select* $\subseteq$ *stat*^{n} be the set of n-tuples of selecting states of a TA A. We can define a query selecting n-tuples of nodes mapped by A to n-tuples of selecting states on some run:

$$Q_\exists(t) = \left\{ (\pi_1, \ldots, \pi_n) \;\middle|\; \begin{array}{l} \text{there is a successful run } r \text{ of } A \text{ on } t \\ \text{where } (r(\pi_1), \ldots, r(\pi_n)) \in \textit{Select} \end{array} \right\}$$

These queries, named *existential run-based queries*, are studied by Niehren et al. in [NPTT05], and proved to capture MSO-definable queries too. Replacing the existential quantification on runs by a universal one does not change their expressiveness. This is no longer the case when considering the class of deterministic automata. The authors also consider unambiguous automata, i.e., automata having at most one accepting run per tree. A property of these automata is that an n-ary query can be defined using an unambiguous automaton iff it can be written as a Boolean combination of monadic MSO formulas. As a consequence, monadic queries defined by unambiguous automata are exactly monadic MSO-definable queries. But for $n > 1$, queries defined by unambiguous automata are strictly less expressive than MSO-definable queries.

In a prior work [FGK03], Frick et al. proposed a monadic variant of this approach, using *selecting tree automata*, and operating on DAGs that are a compact representation of trees. Without compact representation, the query evaluation problem for a selecting tree automaton A on a tree t is in time $O(|A|^3 \cdot |t|)$.

When trees are compressed, the query evaluation problem is in time $2^{O(|A|)} \cdot |t_c|$, where t_c is a compact representation of t. Hosoya and Pierce [HP03] also defined run-based n-ary queries through *pattern automata*, on ranked trees. Neven and Schwentick introduced *query automata* in [NS02]. They start from two-way automata [Mor94] and add selecting states. In terms of expressiveness, *query automata* capture MSO-definable queries. In the unranked case, this is the case only when adding stay transitions. Emptiness, inclusion and equivalence of query automata are all EXPTIME-complete problems.

Other automata models were proposed for processing XML documents in the context of streaming. This led to the introduction of tree automata that run through trees in pre-order traversal of their nodes. We survey such automata in Chapter 4 of this manuscript.

2.3.3 XPath

With the introduction of XML as a standard for semi-structured data [BPSM$^+$08], the W3C defined the XPath query language [CD99]. XPath is used to select sets of nodes in XML documents, based on some properties of paths. XPath is a basis for numerous other standards: XML Schema [FW04] for defining schemas, XPointer [DMJ01] for identifying fragments of XML documents, and XQuery [BCF$^+$07] and XSLT [Cla99] for document transformations.

Two versions of XPath have been released so far. XPath 1.0 defines queries by path expressions, with other features like data value tests, arithmetic operations, aggregators, etc. XPath 2.0 extends XPath 1.0 with the objective of having a first-order complete navigational core, that is missing in XPath 1.0. We present both versions in the following. Known results about expressiveness, evaluation and static analysis of XPath 1.0 are surveyed by Benedikt and Koch in [BK08].

XPath 1.0

XPath 1.0 is a navigational language based on a set of axis related to tree structures. XPath 1.0 expressions define monadic queries using a simple syntax, without variables. Consider for instance the expression: $//cd[author]/title$. It considers all *cd* nodes ($//$ is the descendant, i.e., *ch** axis), tests whether they have an *author* child node ($[\ldots]$ delimits test expressions), and if this is the case, outputs their *title* children nodes ($/$ is the composition of steps, and the default axis is *ch*).

CoreXPath 1.0 As mentioned earlier, XPath 1.0 comes with features that are not navigational. In particular, data value manipulations such as arithmetic operations make XPath 1.0 undecidable. For this reason, Gottlob, Koch and Pichler define CoreXPath 1.0 [GKP05], a formal characterization of the navigational core

axis	d	::=	$\mathit{self} \mid \mathit{foll} \mid \mathit{prec}$
			$\mid \mathit{ch} \mid \mathit{ch}^* \mid \mathit{ch}^+ \mid \mathit{ch}^{-1} \mid (\mathit{ch}^{-1})^* \mid (\mathit{ch}^{-1})^+$
			$\mid \mathit{ns} \mid \mathit{ns}^* \mid \mathit{ns}^+ \mid \mathit{ns}^{-1} \mid (\mathit{ns}^{-1})^* \mid (\mathit{ns}^{-1})^+$
label tests	ℓ	::=	$a \mid *$ (where $a \in \Sigma$)
steps	S	::=	$d::\ell \mid SF$
paths	P	::=	$S \mid S/P$
filters	F	::=	$[P] \mid [not(F)] \mid [F_1 \; and \; F_2]$
rooted path	R	::=	$/P$

Figure 2.5: Syntax of CoreXPath 1.0.

of XPath 1.0. We recall the syntax of CoreXPath 1.0 in Figure 2.5. A CoreXPath 1.0 expression is either a path expression P or a rooted path expression R. CoreXPath 1.0 allows *ns* and ns^{-1} axis, as opposed to the XPath standard.

We progressively define the semantics of each element, when interpreted on a tree $t \in \mathcal{T}_\Sigma$. Label tests and filters are interpreted as unary relations, that select the nodes of the tree satisfying these tests: $[\![.]\!]^t_{\text{filter}} \subseteq \mathit{nod}(t)$. Axis, steps and paths are interpreted as binary relations, relating pairs of nodes of t: $[\![.]\!]^t_{\text{path}} \subseteq \mathit{nod}(t) \times \mathit{nod}(t)$.

The axis *self* relates each node with itself:

$$[\![\mathit{self}]\!]^t_{\text{path}} = \{(\pi, \pi) \mid \pi \in \mathit{nod}(t)\}$$

Axis *ch* and *ns* keep their usual semantics from the definition of t as a relational structure:

$$[\![\mathit{ch}]\!]^t_{\text{path}} = \mathit{ch}^t \qquad [\![\mathit{ns}]\!]^t_{\text{path}} = \mathit{ns}^t$$

We define the transitive and inverse variants of axis using the corresponding operations on binary relations:

$$[\![d^*]\!]^t_{\text{path}} = ([\![d]\!]^t_{\text{path}})^* \qquad [\![d^+]\!]^t_{\text{path}} = ([\![d]\!]^t_{\text{path}})^+ \qquad [\![d^{-1}]\!]^t_{\text{path}} = ([\![d]\!]^t_{\text{path}})^{-1}$$

The *following* (resp. *preceding*) axis relates each nodes with all nodes greater (resp. smaller) than itself in post-order (resp. pre-order) traversal:

$$[\![\mathit{foll}]\!]^t_{\text{path}} = [\![\mathit{ch}^*]\!]^t_{\text{path}} \circ [\![\mathit{ns}^+]\!]^t_{\text{path}} \circ [\![(\mathit{ch}^{-1})^*]\!]^t_{\text{path}}$$
$$[\![\mathit{prec}]\!]^t_{\text{path}} = [\![\mathit{ch}^*]\!]^t_{\text{path}} \circ [\![(\mathit{ns}^{-1})^+]\!]^t_{\text{path}} \circ [\![(\mathit{ch}^{-1})^*]\!]^t_{\text{path}}$$

Label tests have a monadic interpretation, like filters: $[\![\ell]\!]^t_{\text{filter}} \subseteq \mathit{nod}(t)$. The symbol "*" is a wildcard:

$$[\![a]\!]^t_{\text{filter}} = \mathit{lab}^t_a \qquad [\![*]\!]^t_{\text{filter}} = \mathit{nod}(t)$$

A step is a move in the tree along a path, where the target node verifies the label test and the filter test. This is the basic element of a path, and is interpreted as a binary relation:

$$\begin{array}{rcl} [\![d::\ell]\!]^t_{\text{path}} & = & [\![d]\!]^t_{\text{path}} \cap (\mathit{nod}(t) \times [\![\ell]\!]^t_{\text{filter}}) \\ [\![SF]\!]^t_{\text{path}} & = & [\![S]\!]^t_{\text{path}} \cap (\mathit{nod}(t) \times [\![F]\!]^t_{\text{filter}}) \end{array}$$

Finally, path expressions are defined by a series of steps:

$$[\![S/P]\!]^t_{\text{path}} = [\![P]\!]^t_{\text{path}} \circ [\![S]\!]^t_{\text{path}}$$

Path filters are interpreted existentially. Boolean operations are then interpreted as usual:

$$\begin{array}{rcl} [\![[P]]\!]^t_{\text{filter}} & = & \{\pi \mid \exists \pi'.\, (\pi, \pi') \in [\![P]\!]^t_{\text{path}}\} \\ [\![[not(F)]]\!]^t_{\text{filter}} & = & \mathit{nod}(t) - [\![F]\!]^t_{\text{filter}} \\ [\![[F_1 \; and \; F_2]]\!]^t_{\text{filter}} & = & [\![F_1]\!]^t_{\text{filter}} \cap [\![F_2]\!]^t_{\text{filter}} \end{array}$$

Rooted paths are interpreted as the set of nodes accessible when starting at the root node, and following the path. Thus it is a monadic relation: $[\![.]\!]^t_{\text{rpath}} \subseteq \mathit{nod}(t)$:

$$[\![/P]\!]^t_{\text{rpath}} = \{\pi \mid (\epsilon, \pi) \in [\![P]\!]^t_{\text{path}}\}$$

If a CoreXPath 1.0 expression is a path expression P, then it naturally defines the binary query $Q_P(t) = [\![P]\!]^t_{\text{path}}$. If it is a rooted path expression R, it corresponds to the monadic query $Q_R(t) = [\![R]\!]^t_{\text{rpath}}$. The set of binary queries defined by path expressions of CoreXPath 1.0 exactly captures the two variables fragment of FO over unranked trees [MdR05]. This fragment is strictly less expressive than FO. In [Mar05b, Mar05a], Marx shows that any extension of CoreXPath 1.0 closed under path complementation is FO-expressive.

Static Analysis CoreXPath 1.0 is now a well-studied logic. Static problems are analyzed in [NS03, Woo03, MS04, GLS07]. Main results are presented in surveys [GKP03, BK08]. Satisfiability of CoreXPath 1.0 is known to be decidable, even in the presence of DTDs [BFG08]. Containment (also called inclusion) of queries is the problem that takes as input two expressions e and e', and outputs the truth value of $Q_e(t) \subseteq Q_{e'}(t)$ for all $t \in \mathcal{T}_\Sigma$. We write $Q_e \subseteq Q_{e'}$ if this property holds. For binary queries of CoreXPath 1.0, containment is EXPTIME-complete. In this manuscript we will sometimes use reductions to the universality of queries, i.e. given an expression e defining a query, decide whether $\forall t \in \mathcal{T}_\Sigma$, $\forall \tau \in \mathit{nod}(t)^n$, $\tau \in Q_e(t)$.

Proposition 4. *Deciding the universality of Boolean CoreXPath 1.0 filters and monadic CoreXPath 1.0 expressions restricted to axes ch and ch* is coNP-hard.*

Proof. As negation and disjunctions are allowed in CoreXPath 1.0 filters, containment and universality are equivalent, because $\mathcal{T}_\Sigma \subseteq L_{Q_{[not(e_1)\ or\ e_2]}}$ iff $L_{Q_{e_1}} \subseteq L_{Q_{e_2}}$. Moreover, containment of CoreXPath 1.0 filters was proved coNP-hard by Miklau and Suciu [MS04], even for positive filters restricted to axes *ch* and *ch**. In the presence of negation, universality of monadic queries is harder than universality of Boolean queries of the same class. □

For the dynamic approach, we present known query evaluation algorithms in Section 2.3.5.

Forward XPath Forward XPath [Olt07b] is the restriction of CoreXPath 1.0 where allowed axes are only forward axes, i.e. axes d such that if $(\pi, \pi') \in [\![d]\!]^t_{\text{path}}$, then π' follows π in document order. Such axis are:

$$d ::= \textit{self} \mid \textit{foll} \mid \textit{ch} \mid \textit{ch}^* \mid \textit{ch}^+ \mid \textit{ns} \mid \textit{ns}^* \mid \textit{ns}^+$$

This set of axis is often used for streaming XML: matches of Forward XPath expressions can be built progressively along the stream, without guessing unread information. This restriction on axes does not affect expressiveness: every CoreXPath 1.0 expression can be rewritten into an equivalent Forward XPath expression [OMFB02]. However this translation can produce exponentially bigger expressions.

CoreXPath 1.0 Extensions Some extensions of CoreXPath 1.0, inspired by temporal logics, were proposed. For example, Marx defines *Conditional XPath* [Mar04a, Mar05a] from CoreXPath 1.0 by adding path expressions of the form $(SF)^+$ where $S = d::\ell$ is a step and F a filter expression. This expression is interpreted as the transitive closure: $[\![(SF)^+]\!]^t_{\text{path}} = ([\![SF]\!]^t_{\text{path}})^+$, i.e., we can move according to S, and at each step we must check that F is true. This is inspired by the *Until* operator of temporal logics: we can do jumps along S until some position, and on the way F is true at each step. Conditional XPath is FO-complete, and thus strictly more expressive than CoreXPath 1.0. As we will see later on, this does not increase the evaluation time.

Beyond Conditional XPath, *Regular XPath* [Mar04b] allows transitive closure of any path expression, not only steps. In [tC06], ten Cate defines Regular XPath$^\approx$ as the extension of Regular XPath by the equality operator $\approx$. Given two path expressions P_1 and P_2, $P_1 \approx P_2$ is true at node π of t if there is a node π' that can be reached from π by both P_1 and P_2. It is still unknown whether this operator is needed. In terms of expressiveness, Regular XPath$^\approx$ (when considered as a binary query language) is a strict extension of Conditional XPath, as it captures FO*, the FO logic over trees allowing a transitive closure operator on formulas having exactly two free variables. However, Regular XPath does not capture

MSO-definable queries. Indeed, ten Cate and Segoufin recently proved that FO* is strictly included in MSO for trees [tCS08] (they proved a more general result, as their transitive closure operator allows for more than two free variables). The evaluation of a Regular XPath expression e on a tree t can be performed in time $O(|t| \cdot |e|)$ [Mar04b].

CoreXPath 1.0 with (attribute) data value comparisons has also been studied. Its satisfiability is undecidable in the general case [GF05], but becomes decidable with restriction on allowed axes [BDM$^+$06, BFG08, Fig09]. In particular, horizontal axes introduce additional difficulties [GF05, BFG08]. In [Par09], Parys proves that CoreXPath 1.0 expressions e with data value comparisons can be evaluated in time $O(|t| \cdot |e|^3)$. Adding aggregators leads to an exponential blow-up in the query size.

Tree Patterns

Tree patterns are similar to CoreXPath 1.0 queries using only descending axis *ch* and *ch**, and no negation and disjunction. They define n-ary queries using variables $\mathcal{V}_n = \{x_1, \ldots, x_n\}$. Tree patterns are expressions of the form $/F$ where F is defined by the following grammar:

$$F ::= \mathit{and}(F_1, F_2) \mid \mathit{ch}(F) \mid \mathit{ch}^*(F) \mid \ell(F) \mid x \mid \mathit{true}$$

where $\ell \in \Sigma$, $x \in \mathcal{V}_n$, $d \in \{\mathit{ch}, \mathit{ch}^*\}$, and the operator $/$ appears in root position only. The semantic $[\![F]\!]_{t,\mu} \subseteq \mathit{nod}(t)$ is defined modulo an assignment $\mu : \mathcal{V}_n \to \mathit{nod}(t)$ and the following equations:

$$\begin{aligned}
[\![\mathit{and}(F_1, F_2)]\!]_{t,\mu} &= [\![F_1]\!]_{t,\mu} \cap [\![F_2]\!]_{t,\mu} \\
[\![\mathit{ch}(F)]\!]_{t,\mu} &= \{\pi \mid \exists \pi' \in [\![F]\!]_{t,\mu}.\ \mathit{ch}^t(\pi, \pi')\} \\
[\![\mathit{ch}^*(F)]\!]_{t,\mu} &= \{\pi \mid \exists \pi' \in [\![F]\!]_{t,\mu}.\ (\mathit{ch}^*)^t(\pi, \pi')\} \\
[\![\ell(F)]\!]_{t,\mu} &= \{\pi \mid \ell = \mathit{lab}^t(\pi)\} \\
[\![x]\!]_{t,\mu} &= \{\mu(x)\} \\
[\![\mathit{true}]\!]_{t,\mu} &= \{\mathit{nod}(t)\} \\
[\![/F]\!]_{t,\mu} &= \{\epsilon\} \cap [\![F]\!]_{t,\mu}
\end{aligned}$$

The query defined by a tree pattern $/F$ is given by:

$$Q(t) = \{(\mu(x_1), \ldots, \mu(x_n)) \mid \epsilon \in [\![/F]\!]_{t,\mu}\}$$

Sometimes [BKS02], the query is composed by the matchings of all nodes of the expression, i.e., for each step a new variable is present. Miklau and Suciu [MS04] show that inclusion of tree patterns is coNP-complete. In [BFK05], Benedikt et al. study the sublanguages of XPath obtained by removing filters, downward recursion, and/or upward axis, while never allowing horizontal axis. They relate these fragments to tree patterns, in terms of expressiveness.

axis	d	::=	*self* | *foll* | *prec*
			| *ch* | *ch** | *ch*$^+$ | *ch*$^{-1}$ | (*ch*$^{-1}$)* | (*ch*$^{-1}$)$^+$
			| *ns* | *ns** | *ns*$^+$ | *ns*$^{-1}$ | (*ns*$^{-1}$)* | (*ns*$^{-1}$)$^+$
label tests	ℓ	::=	a | $*$ (where $a \in \Sigma$)
steps	*S*	::=	d::ℓ | *SF*
node test	*N*	::=	. | x (where $x \in \mathcal{V}$)
paths	*P*	::=	*S* | *S*/*P*
			| *P*$_1$ *union* *P*$_2$ | *P*$_1$ *intersect* *P*$_2$ | *P*$_1$ *except* *P*$_2$
			| *N* | *for* x *in* *P*$_1$ *return* *P*$_2$
filters	*F*	::=	[*P*] | [not(*F*)] | [*F*$_1$ and *F*$_2$] | [*N*$_1$ *is* *N*$_2$]
rooted path	R	::=	/*P*

Figure 2.6: Syntax of CoreXPath 2.0.

XPath 2.0

XPath 2.0 [KRS$^+$07] has been defined from XPath 1.0 by adding some features, in order to get a more expressive query language. XPath 2.0 permits the use of variables x (from an infinite set $\mathcal{V}$ of variables). These are interpreted as path expressions that move from any node to the node assigned to x. A test is added to compare nodes assigned to variables: [x *is* .] tests whether the current node is the one assigned to x, whereas [x *is* y] is true if x and y are both assigned to the current node. An iterator is also added, through for-loops of the form *for* x *in* P_1 *return* P_2. This is interpreted as a path expression. Two path expressions operators are also added: the relative complement *P*$_1$ *except* *P*$_2$, the union *P*$_1$ *union* *P*$_2$ and the intersection *P*$_1$ *intersect* *P*$_2$.

CoreXPath 2.0 is a formalization of the navigational core of XPath 2.0 proposed by ten Cate and Marx [tCM07]. Its syntax is detailed in Figure 2.6.

We define only the semantics of the new elements of CoreXPath 2.0, as elements coming from CoreXPath 1.0 keep the same semantics. More precisely, the semantics of a CoreXPath 2.0 expression e on a tree t is done modulo an assignment μ of the free variables of e to nodes of t. CoreXPath 1.0 expressions only propagate this assignment, whereas CoreXPath 2.0 expressions use it in the

following way:

$$\begin{aligned}
[\![.]\!]^{t,\mu}_{\text{path}} &= [\![\textit{self}]\!]^{t,\mu}_{\text{path}} \\
[\![x]\!]^{t,\mu}_{\text{path}} &= \textit{nod}(t) \times \{\mu(x)\} \\
[\![P_1 \textit{ union } P_2]\!]^{t,\mu}_{\text{path}} &= [\![P_1]\!]^{t,\mu}_{\text{path}} \cup [\![P_2]\!]^{t,\mu}_{\text{path}} \\
[\![P_1 \textit{ intersect } P_2]\!]^{t,\mu}_{\text{path}} &= [\![P_1]\!]^{t,\mu}_{\text{path}} \cap [\![P_2]\!]^{t,\mu}_{\text{path}} \\
[\![P_1 \textit{ except } P_2]\!]^{t,\mu}_{\text{path}} &= [\![P_1]\!]^{t,\mu}_{\text{path}} - [\![P_2]\!]^{t,\mu}_{\text{path}} \\
[\![\textit{for } x \textit{ in } P_1 \textit{ return } P_2]\!]^{t,\mu}_{\text{path}} &= \{(\pi_1,\pi_2) \mid \exists \pi_3 \in \textit{nod}(t). \\
&\qquad (\pi_1,\pi_3) \in [\![P_1]\!]^{t,\mu}_{\text{path}} \text{ and } (\pi_1,\pi_2) \in [\![P_2]\!]^{t,\mu[x\leftarrow\pi_3]}_{\text{path}}\} \\
[\![[. \textit{ is } x]]\!]^{t,\mu}_{\text{filter}} &= \{\mu(x)\} \\
[\![[. \textit{ is } .]]\!]^{t,\mu}_{\text{filter}} &= \textit{nod}(t) \\
[\![[x \textit{ is } y]]\!]^{t,\mu}_{\text{filter}} &= [\![. \textit{ is } x]\!]^{t,\mu}_{\text{filter}} \cap [\![. \textit{ is } y]\!]^{t,\mu}_{\text{filter}}
\end{aligned}$$

This time, CoreXPath 2.0 path expressions *P* (and similarly for rooted path expressions R) define n-ary queries by the assignments that satisfy the expression:

$$Q_P(t) = \{(\pi_1,\ldots,\pi_n) \mid [\![P]\!]^{t,[x_1\leftarrow\pi_1,\ldots,x_n\leftarrow\pi_n]}_{\text{path}} \neq \emptyset\}$$

The problem of query inclusion for various fragments of CoreXPath 2.0 is studied in [tCL07]. It ranges from EXPTIME (for the extension of CoreXPath 1.0 with path equality) and 2-EXPTIME (for the extension with path intersection), to non-elementary (for the extension with path complementation or for-loops). The equivalence problem is shown decidable in [tCM07]. Satisfiability of XPath 2.0 was studied in [Hid03] before the axomatization of XPath 2.0 by CoreXPath 2.0. In terms of expressiveness, CoreXPath 2.0 is FO-complete. Some FO-expressive fragments of CoreXPath 2.0 enjoying efficient evaluation algorithms are presented in [FNTT07]. We present them in Section 2.3.5.

2.3.4 Other Approaches for Querying in Trees

In this section we briefly survey some other formalisms proposed for querying finite ordered trees.

Conjunctive Queries

A *conjunctive query* $Q(x_1,\ldots,x_n)$ over a signature $\Delta = \{r_1,\ldots,r_k\}$ is a FO[Δ] formula only using conjunctions, and existential quantifiers at the outermost levels, as for instance:

$$\phi(x_1) = \exists y_1.\, \exists y_2.\, r_1(x_1,y_1) \;\wedge\; r_2(y_2)$$

Conjunctive queries enjoy a clean relation with the Project/Join algebra, and thus are also studied in the context of relational databases [AHV95].

Conjunctive queries over trees are studied by Gottlob et al. in [GKS06]. The authors investigate the tractability of the query evaluation problem, depending on which XPath axis are used in the signature Δ. A frontier is established for arbitrary finite structures, and then applied to XPath axis. Depending on the chosen set of XPath axis, the query evaluation is either in PTIME or NP-hard. In [BFLS06], Bry et al. investigate algorithms for conjunctive n-ary queries over graphs, that are also efficient on trees.

Other restrictions over conjunctive queries are studied in the context of relational databases. For instance *acyclic conjunctive queries* are introduced by Yannakakis [Yan81]. These are conjunctive queries which corresponding hypergraph representation is acyclic. Yannakakis proposed an algorithm that evaluates these queries Q in time $O(|D| \cdot |\phi| \cdot |\phi(D)|)$ for a database D. Some algorithms for evaluating acyclic conjunctive queries incrementally are proposed by Bagan et al. [BDG07]. *Tree patterns* (as presented in Section 2.3.3) are a special case of acyclic conjunctive queries on tree structures.

Monadic Datalog

Datalog *Datalog* is a generalization of conjunctive queries, introducing recursion. A Datalog program is a set of Datalog rules, each of them being composed by a head (an atom) and a body (a conjunction of atoms, i.e. a conjunctive query). For instance the conjunctive query mentioned in the preceding paragraph corresponds to the rule:

$$\phi(x_1) \text{ :- } r_1(x_1, y_1),\ r_2(y_2).$$

Datalog comes with the least fixed point semantics, as explained below for ground Datalog. For precise definitions and results, see for instance [AHV95, CGT90].

Monadic Datalog In [GK04], Gottlob and Koch propose *Monadic Datalog* as a monadic query language over unranked trees. A Monadic Datalog program is a Datalog program where all head predicates are unary, and one of these is considered as the selecting predicate, thus defining a monadic query. Gottlob and Koch consider the signature $\Lambda = \{\mathit{fc}, \mathit{ns}, \mathit{root}, \mathit{leaf}, \mathit{lc}\} \cup \{\mathit{lab}_a \mid a \in \Sigma\}$, where *root, leaf* and *lc* are monadic predicates respectively selecting the root node, the leaves and the last children (i.e., children nodes without next-sibling). Over this signature, Monadic Datalog programs exactly capture monadic queries that are MSO[*fc*, *ns*]-definable. The query evaluation of a Monadic Datalog program P on a tree t is in linear combined complexity: $O(|t| \cdot |P|)$.

Ground Datalog In this manuscript we sometimes use *ground Datalog* as a simple way to define new relations. A ground Datalog program is a Datalog pro-

gram without variables. We recall here the definition, and the key result about the linear resolution of such programs.

Let Λ be a ranked signature containing constants $c \in \Lambda$ and predicates $p \in \Lambda$, where all predicates have an arity $ar(p) \in \mathbb{N}_0$. We call a term $p(c_1, \ldots, c_{ar(p)})$ a *literal*, and denote the set of literals over Λ by $lit(\Lambda)$. A *clause* is a pair in $lit(\Lambda) \times lit(\Lambda)^k$ (with $k \in \mathbb{N}_0$) that we write L :- $L_1, \ldots, L_k$. as usual. A *ground Datalog program* P is a finite set of clauses over Λ. Its size $|P|$ is the total number of symbols appearing in all its clauses.

The *least fixed point* $lfp(P)$ of P is the least set of literals over Λ that satisfies that for all clauses L :- $L_1, \ldots, L_k$. of P, if $L_1, \ldots, L_k \in lfp(P)$ then $L \in lfp(P)$. As no negation is allowed, every ground Datalog program P has a unique least fixed point, and this one is finite. For ground Datalog, this least fixed point can be efficiently computed [CGT89, DEGV01, GGV02].

Proposition 5. *For every signature Λ and every ground Datalog program P over Λ, the least fixed point of P can be computed in time $O(|P|)$.*

Modal Logics

Modal logics are logics using modality operators. Among these logics, temporal logics are a popular way to describe properties of dynamic systems, and check them by verification techniques. They can be used to express that a property will be satisfied in some system continuation, in all continuations, or to check that a property is true until some time point where another property is true. In trees, properties are expressed on paths of the tree. We briefly mention some works on temporal logics over ordered trees (see [Lib06] for a more complete overview).

Linear Temporal Logic (LTL) is known to capture FO on words, by Kamp's Theorem [Kam68]. In [Mar05a], Marx adapts the definition of LTL to trees by using two variants for each modality operator: one for horizontal paths (along *ns*), one for vertical paths (along *ch*). The resulting logic is equivalent to FO$[ch^*, ns^*]$, in terms of expressiveness, for Boolean and unary queries. Benedikt and Jeffrey [BJ07] consider the *Hennessy-Milner Logic* (HML) [HM85], obtained from the previous logic by lifting the *until* modality. This way, they capture CoreXPath 1.0.

Computation Tree Logic (CTL) and CTL* add branching to the LTL approach, by distinguishing node formulas and path formulas (in the same way as XPath uses filters and path expressions). CTL* was proved equivalent to FO for binary trees for a long time [HT87], and recently Barceló and Libkin proved that CTL^*_{past} is equivalent to FO over unranked trees [BL05]. "*past*" means here that ch^{-1} and ns^{-1} are also used in modality operators.

Propositional Dynamic Logic (PDL) has also been adapted to trees by Afanasiev et al. [ABD$^+$05]. PDL_{tree}, the resulting logic, is based on Boolean

combinations (and existential quantification) of path formulas where branching and transitive closures are allowed. Its expressiveness is exactly the same as Regular XPath [Mar04b].

The *modal μ-calculus* adds least and greatest fixed points to modal logics. Barceló and Libkin studied this logic in the context of unranked trees [BL05]. For Boolean and monadic queries, the μ-calculus based on axis *fc* and *ns* is equivalent to MSO. Some logics inspired from the μ-calculus were later defined [GLS07] to improve the satisfiability checking.

Other Models of Queries

We briefly mention other formalisms for querying in trees.

Neumann and Seidl define monadic queries by *forest grammars* [NS98], that were extended to n-ary queries by Berlea and Seidl [BS04]. In order to evaluate these queries, Neumann and Seidl introduce *pushdown forest automata*. These automata traverse the input tree in pre-order, and thus permit a streaming evaluation. For this reason, we present this work in more details in Chapter 3.

Regular path queries are queries on graphs, defined by regular expressions on basic steps (like XPath steps) [ABS00]. In trees, this corresponds to *caterpillar expressions*, as defined by Brüggemann-Klein and Wood [BKW00]. These are strictly less expressive than MSO, and incomparable with FO. Goris and Marx define *looping caterpillars* [GM05] by adding a loop predicate, that only keeps loops of an expression. Looping caterpillar are able to capture binary FO queries on unranked trees.

Regular expressions can also be used at a higher level, to define *regular expression patterns*. In [BCF03], Benzaken et al. propose *CDuce*, a typed programming language for XML. This language uses such regular expression patterns to select hedge elements. These patterns are based on tree variables, hedge algebra operators, and regular expressions operators. Here, a syntactic restriction avoids subtree equality tests. These are allowed in the more general Tree Query Logic [CG04, FTT07], a spatial logic for ordered trees.

Some work has also be done for combining existing query formalisms. In particular, Boolean and monadic queries can be used to define n-ary queries, as explained for instance in [Sch00, NS00, FNTT06, ABL07].

2.3.5 Evaluation Algorithms

In this section we survey the complexity of outputting all the answers of a query, for the different classes related to our framework. We survey results for algorithms without streaming constraints (see also the survey by Koch [Koc06]). The related work on streaming is in Chapter 1.

Query Evaluation and Enumeration

We present two frameworks for computing answers of a query.

1. Query *evaluation* is the more general framework, that measures the overall time required to output the set of all answers.

2. Query *enumeration* [JPY88, GS03a, Bag06, Cou09] distinguishes the precomputation and the delay between consecutive answers. Hence the first answer can usually be output more quickly than by computing the whole answer set.

These frameworks do not take space complexity into account, as the tree is entirely stored in main memory. More precisely, in the enumeration framework, space and time are bounded by the same function during the incremental computation of answers, but no restriction is made during the preprocessing phase [Bag09]. We provide the definitions in the sequel. Let $\mathcal{Q}$ be a query class, each expression $e \in \mathcal{Q}$ being equipped with a size $|e| \in \mathbb{N}_0$ and defining a query Q_e.

We say that $\mathcal{Q}$ can be *evaluated* in time f, if there exists an algorithm that takes as input any expression $e \in \mathcal{Q}$ and any tree $t \in \mathcal{T}_\Sigma$, and outputs the set $Q_e(t)$ in time less than $O(f(|t|, |e|, |Q_e(t)|))$, where $|Q_e(t)|$ is the number of elements in $Q_e(t)$. Note that query evaluation is harder than satisfiability.

The class $\mathcal{Q}$ can be *enumerated* with preprocessing f and delay d if there exists an algorithm that takes as input any expression $e \in \mathcal{Q}$ and tree $t \in \mathcal{T}_\Sigma$, has a preprocessing phase of time less than $O(f(|t|, |e|))$, and then enumerates all the answers $Q_e(t)$ with a delay at most $d(|t|, |e|)$ between two consecutive answers. There is no restriction on the output order of answers. Outputting an answer twice is forbidden.

Query enumeration is an intermediate model between the standard evaluation and the streaming evaluation. It is a special case of query evaluation algorithms, while streaming query answering algorithms can be considered as special cases of enumeration algorithms, with the additional constraint on the traversal order, and with a focus on space consumption.

A recent work by Bagan et al. introduces two other frameworks [BDGO08]. The first one is the computation of a random solution, whereas the second one is the computation of the j-th solution. Another problem is to maintain the set of answers while the XML document is updated. This is usually referred as the *view maintenance problem* [SI84, GMS93, BGMM09].

Automata, FO and MSO defined Queries

The evaluation of FO formulas over relational structures is PSPACE-complete. Once the query is fixed, it becomes a PTIME problem [Var95].

In [DO06], Durand and Olive study the enumeration complexity for queries defined by first-order formulas on quasi-unary structures. Quasi-unary structures are structures s over a signature Δ containing unary relations symbols, plus one function $f\colon \mathit{dom}(s) \to \mathit{dom}(s)$. In particular labeled unordered unranked trees can be encoded into quasi-unary structures. They prove that enumeration over these structures can be done with a precomputation linear in the size of the structure and the query, and a delay linear in the size of the query (independent of the structure size).

Satisfiability, and thus evaluation, of MSO formulas is non-elementary. Once more, this is not the case when the formula is fixed. In [Bag06], Bagan provides an enumeration algorithm that progressively outputs answers of any query defined by an MSO formula over trees (in fact, over the more general class of graphs of bounded tree-width). This algorithm avoids duplicate answers, has a precomputation phase linear in $|t|$ and a delay linear in the arity n, when the formula $\phi(x_1, \ldots, x_n)$ is fixed.

For queries defined by automata, Bagan also proposes in [Bag06] an algorithm with a precomputation time in $O(|A|^3 \cdot |t|)$ where A is an automaton recognizing the canonical language of the query (with universal schema). Its delay between answers is in $O(n)$, where n is the arity.

XPath

The first XPath query engines were known to use exponential time, even for CoreXPath 1.0 queries. In [GKP03, GKP05], Gottlob et al. propose an algorithm that evaluates the full XPath 1.0 language in PTIME combined complexity (i.e., polynomial in both expression $|e|$ and XML document size $|t|$). Moreover, this algorithm runs in linear combined complexity $O(|t| \cdot |e|)$ for CoreXPath 1.0 queries. The algorithm is simply based on a bottom-up semantic of XPath. By other means, Ramanan proves the same result on the positive fragment of CoreXPath 1.0 [Ram03]. Marx showed that the evaluation of Conditional XPath and Regular XPath also enjoys PTIME combined complexity [Mar04b]. In terms of data complexity Gottlob et al. show in [GKPS05] that the query evaluation problem (and validation) is not PTIME-hard, but belongs to lower (parallelizable) complexity classes. Marian and Siméon [MS03] propose a projection technique, such that useless parts of the XML document (w.r.t. to a given query) are not loaded in main memory.

CoreXPath 2.0 is known to capture FO-definable n-ary queries modulo linear time transformations. As a consequence, the evaluation is PSPACE-complete for CoreXPath 2.0, and no PTIME algorithm exists unless PTIME=PSPACE. In [FNTT07], Filiot et al. exhibit a fragment of CoreXPath 2.0, that enjoys a PTIME evaluation, while still being FO-complete. This fragment imposes the following

restrictions: no quantifiers, no variable sharing in path composition, and no variables below complementation. To the best of our knowledge, there are no results for the enumeration of XPath queries.

Tree Patterns

Many algorithms were proposed for evaluating tree patterns, a subclass of Core-XPath 1.0. The first algorithms evaluating tree patterns (also called *twig* patterns) computed all pairs of nodes satisfying each step of the query, and then joined them to output the answers. This approach computes a lot of useless intermediate results. A first improvement, named *TwigStack*, was proposed by Bruno et al. [BKS02]. It is based on a technique named *holistic twig join*, that checks for matchings along a root-to-leaf path, instead of steps. However, the algorithm still computes too much intermediate results (more than the size of the answer set) in presence of child axis.

Some improvements were subsequently proposed. Jiang et al. [JWLY03] eliminate more intermediate matchings, while Chen [Che06] improves their merging. Chen et al. propose *Twig2Stack* [CLT$^+$06]. Their algorithm deals with *Generalized Tree Patterns*, i.e., tree patterns that allow *for*-loops *à la* XPath 2.0. Their algorithm runs in time $O(|t| \cdot |e|)$ for usual tree patterns e. Some further improvements were presented in [ZXM07, JLH$^+$07]. We refer the reader to [GC07b] for a more complete survey on tree patterns.

Validation

In [Seg03], Segoufin proves that the validation problem ranges from LOGSPACE-complete to LOGCFL-complete, depending on the schema language and representation (this includes DTDs and EDTDs). Martens et al. [MNSB06b] study the more specific case of XML Schema, but mostly in terms of expressiveness.

Chapter 3

Streamability

Contents

3.1 Introduction

Query answering in streaming mode is a challenging issue. Streaming evaluation aims for low memory consumption. However, most of query languages, like the W3C language XPath, are not designed for streaming evaluation. A measure for the difficult of a query for streaming processing is its *concurrency*. The concurrency of a query is the maximal number of simultaneous candidate solutions, that can be selected or not, depending on the end of stream. Concurrency was introduced by Bar-Yossef et al., and proved to be a lower memory bound for fragments of XPath [BYFJ05]. Unfortunately, XPath expressions may have unbounded concurrency, such as for instance $/ch^*::*$.

In this chapter, we present our definition of query answering over XML streams. We start with the correspondence between XML documents and their serialization, i.e. the linearization of trees. We propose a computational model named *Streaming Random Access Machines* (SRAMs) in order to formally define the intended inputs and outputs of streaming query answering algorithms, and the corresponding complexity measure. We define the complexity of SRAMs in terms of space and time, in order to study the relationship between efficient buffering and computational cost. In particular, we prove in Chapter 5 some hardness results for time complexity, when only alive candidates are buffered.

We propose a measure of *streamability* for query classes. Roughly speaking, for $m \in \mathbb{N}_0 \cup \{\infty\}$, m-streamable queries can be processed in polynomial space and time when evaluated on trees inducing concurrency less than m. This definition generates a hierarchy of query classes. We investigate the characteristics of this hierarchy, and show which properties must be verified by a query class in order to be ∞-streamable, the queries that are most suitable to streaming in our hierarchy.

Finally, we prove hardness results for testing bounded concurrency for a query class. We also show the consequence of being streamable, and apply these results on XPath. For Forward XPath, we get negative results: deciding bounded concurrency is coNP-hard, and Forward XPath is not m-streamable, for all $m \in \mathbb{N} \cup \{\infty\}$.[1] This motivates further investigations on streamable fragments of Forward XPath.

Other computational models were already proposed for stream processing of XML documents. In [SV02], Segoufin and Vianu study the validation of XML documents in a streaming mode, with bounded memory. In this case, requiring bounded memory is equivalent to the existence of a finite state automaton (without stack) recognizing the language of valid trees. More elaborated machines for stream processing were proposed by Grohe, Koch and Schweikardt [GKS07, Sch07a]. Their machine model uses external memory to measure buffering requirements of algorithms, and allows to read the input stream several times. They infer tight bounds for the complexity of evaluating CoreXPath 1.0 queries over XML streams, in the Boolean and monadic cases. When restricted to a single scan of the input stream, they prove that the depth of the corresponding tree is a lower memory bound, for monadic CoreXPath 1.0 expressions. Benedikt and Jeffrey [BJ07] proposed a simpler model based on Turing machines. They define tractable query classes for this model. We show in this chapter that two of these are ∞-streamable according to our model.

[1] We proved stronger hardness results in follow-up work.

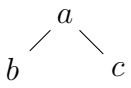
3.2 Streaming

We start this section with a description of XML streams and the definition of our computational model for evaluating queries in XML streams. We formally introduce the notion of concurrency, that we will use later on to define our streamability measure.

3.2.1 Linearizations of Trees

A streaming algorithm that answers a query Q for some class of structures $\mathcal{S}$ reads a linearization of a structure $s \in \mathcal{S}$ from the input stream, and computes a collection of answers $Q(s)$ incrementally. For words, linearization is straightforward, as words are already linear data structures.

Unranked trees need linearization in order to be put onto a stream. For every set S, let



$$\widehat{S} = \{\mathit{op}, \mathit{cl}\} \times S$$

$\widehat{\Sigma}$ is the set of tagged opening and closing parenthesis. An opening parenthesis (op, a) corresponds to the XML tag `<a>` and a closing parenthesis (cl, a) to the XML tag `</a>`. For every tree $t \in \mathcal{T}_\Sigma$ we define the *visible word* $\mathrm{vw}(t) \in \widehat{\Sigma}$ by linearization as follows:

$$\mathrm{vw}(a(t_1, \ldots, t_n)) = (\mathit{op}, a) \cdot \mathrm{vw}(t_1) \cdot \ldots \cdot \mathrm{vw}(t_n) \cdot (\mathit{cl}, a)$$

This word is well-nested in that every opening parenthesis is properly closed. The letters of the visible word $\mathrm{vw}(t)$ can be identified with elements of the following set:

$$\mathit{eve}(t) = \{\mathit{start}\} \cup \widehat{\mathit{nod}(t)}$$

We illustrate the definitions at the tree $t = a(b, c)$. The XML stream for t, its corresponding visible words $\mathrm{vw}(t)$ and its set of events are as follows:

XML stream	`<a> <b> </b> <c> </c> </a>`
$\mathrm{vw}(t) =$	$(\mathit{op}, a) \cdot (\mathit{op}, b) \cdot (\mathit{cl}, b) \cdot (\mathit{op}, c) \cdot (\mathit{cl}, c) \cdot (\mathit{cl}, a)$
$\mathit{eve}(t) =$	$\{\mathit{start}, (\mathit{op}, \epsilon), (\mathit{op}, 1), (\mathit{cl}, 1), (\mathit{op}, 2), (\mathit{cl}, 2), (\mathit{cl}, \epsilon)\}$

Let $\preceq$ be the total order on $\mathit{eve}(t)$ corresponding to the total order of $\mathit{pos}(\mathrm{vw}(t))$ and $\mathit{pr}(e) \in \mathit{eve}(t)$ be the immediate predecessor of an event $\eta \in \widehat{\mathit{nod}(t)}$. For instance, $\mathit{pr}((\mathit{op}, 2)) = (\mathit{cl}, 1)$ in our example. We write

$$\mathit{dom}_\eta(t) = \{\pi \in \mathit{nod}(t) \mid (\mathit{op}, \pi) \preceq \eta\}$$

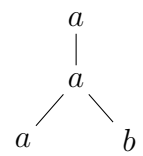
for the set of all nodes visited until event η.

We extend the definitions to hedges, in a straightforward manner. A hedge $h \in \mathcal{H}_\Sigma$ has the following set of events: $\mathit{eve}(h) = \mathit{start} \cup \widehat{\mathit{nod}(h)}$. For $h = (t_1, \ldots, t_k)$, the order $\preceq$ is a total order on $\mathit{eve}(h)$, where *start* is the least event, the events of t_i ($1 \leq i \leq k$) are ordered according to the previous definition for trees, and events of t_i are all inferior to those of t_j, if $i < j$.

3.2.2 Example of Stream Processing

Before defining our computational model, we provide an example for streaming query evaluation.

Consider the monadic query Q_0 that selects all nodes labeled by a and having a b child. This corresponds to the XPath expression: $/\mathit{ch}^*::a[\mathit{ch}::b]$. We suppose here that the domain of Q_0 is $\mathcal{T}_\Sigma$. Let $t_0 = a(a(a, b))$ as illustrated on the right. In the following table, we present the run of a streaming algorithm computing $Q(t_0)$ incrementally.



input	<a>	<a>	<a>					
buffer	ϵ	ϵ 1	ϵ 1 1·1	ϵ 1				
output				$\{\epsilon, 1\}$				

When an a-node is read, it is buffered as we have to wait for a b-child in order to decide for its selection, or to wait until closing time in case of rejection. Here, only nodes ϵ and 1 are selected, and we can observe that they are output exactly when a b child is opened, and thus at the earliest time point. Similarly, the candidate $1{\cdot}1$ is rejected exactly when closing this node, and it could not be rejected before.

For n-ary queries, the output is a set of tuples of nodes. Hence, the buffered candidates are also tuples, that can be partial, as some components might not be known yet. We use the symbol $\bullet$ to mark these components. Consider for instance the binary query Q_1 without schema defined by the XPath 2.0 expression $/\mathit{ch}^*::a[x_1][\mathit{ns}::b[x_2]]$. Q_1 selects all pairs (π_a, π_b), where π_a is labeled by a, π_b is labeled by b, and π_b is the next sibling of π_a, i.e. $\mathit{ns}(\pi_a, \pi_b)$. The run of an algorithm computing Q_1 on the tree t_0 is for instance:

input	<a>	<a>	<a>					
buffer	$(\epsilon, \bullet)$	$(\epsilon, \bullet)$ $(1, \bullet)$	$(\epsilon, \bullet)$ $(1, \bullet)$ $(1{\cdot}1, \bullet)$	$(\epsilon, \bullet)$ $(1, \bullet)$ $(1{\cdot}1, \bullet)$	$(\epsilon, \bullet)$ $(1, \bullet)$ $(1{\cdot}1, 1{\cdot}2)$	$(\epsilon, \bullet)$ $(1, \bullet)$ $(1{\cdot}1, 1{\cdot}2)$	$(\epsilon, \bullet)$ $(1, \bullet)$	$(\epsilon, \bullet)$ $(1, \bullet)$
output							$\{(1{\cdot}1, 1{\cdot}2)\}$	

Here the algorithm chooses to output the answers (π_a, π_b) at the parent node of π_b. This provides a time-efficient algorithm, as we are sure at this time point to have enough information to decide for selection. However this implies to buffer candidates longer than required. For instance here the answer pair $(1 \cdot 1, 1 \cdot 2)$ may be output when opening node $1 \cdot 2$. We study the time cost for achieving such earliest selection (and rejection) in Chapter 5. Note also that adding schema information can improve buffering. For instance if Q_1 had a schema where only a-nodes having two a-ancestors can have a b-sibling, then the two first candidates could have been rejected immediately.

3.2.3 Concurrency

We define the notion of *concurrency*, that intuitively captures the number of candidates to be buffered simultaneously, as proposed by Bar-Yossef et al. [BYFJ05]. This is a key notion for lower bounds in memory consumption. We will use it in the definition of our computational model, and our streamability measure.

Prefix Tree For every event $\eta \in \widehat{\mathit{nod}(t)}$, let the prefix tree $t^{\preceq\eta}$ be the fragment of t which contains all nodes of t opened before (and including) η: $\mathit{nod}(t^{\preceq\eta}) = \mathit{dom}_\eta(t)$, and satisfying $\mathit{lab}^{t^{\preceq\eta}}(\pi) = \mathit{lab}^t(\pi)$ for all $\pi \in \mathit{nod}(t^{\preceq\eta})$. Note that $t^{\preceq(cl,\pi)}$ contains all proper descendants of π in t, while $t^{\preceq(op,\pi)}$ does not. For two trees $t, t' \in \mathcal{T}_\Sigma$ and $\eta \in \mathit{eve}(t)$ we define the predicate $\mathit{equal}_\eta(t, t')$, that holds if t and t' have the same prefix until η:

$$\mathit{equal}_\eta(t, t') \quad \text{iff} \quad \eta \in \mathit{eve}(t) \cap \mathit{eve}(t') \text{ and } t^{\preceq\eta} = t'^{\preceq\eta}$$

Partial Candidates As already mentioned in the previous example, partial candidates τ are elements of $\mathit{dom}^\bullet_\eta(t)^n$ where:

$$\mathit{dom}^\bullet_\eta(t) = \mathit{dom}_\eta(t) \uplus \{\bullet\}$$

The symbol $\bullet$ denotes components where no selection occurred so far. Completions $\mathit{compl}(\tau, t, \eta)$ are complete candidates obtained by replacing $\bullet$-components of τ by nodes of t opened after η:

$$\mathit{compl}((\pi_1, \ldots, \pi_n), t, \eta) = \left\{ (\pi'_1, \ldots, \pi'_n) \in \mathit{nod}(t)^n \;\middle|\; \begin{array}{l} \text{for all } 1 \leq i \leq n, \\ \pi_i \neq \pi'_i \Rightarrow \\ \pi_i = \bullet \wedge \eta \prec (\mathit{op}, \pi'_i) \end{array} \right\}$$

We call a candidate *complete* if it does not contain $\bullet$-components.

Alive Candidates and Concurrency Let Q be an n-ary query. We call a candidate τ *alive* at event η of a tree t, if the information in $t^{\preceq\eta}$ is not sufficient for selection or rejection of τ, i.e., if there exists a continuation of t after η that selects (a completion of) this candidate, and another one that does not:

$$(\tau,\eta) \in \mathit{alive}_Q(t) \Leftrightarrow$$
$$\begin{cases} \quad \exists t' \in \mathit{dom}(Q).\, \mathit{equal}_\eta(t,t') \wedge \exists \tau' \in \mathit{compl}(\tau,t',\eta).\ \tau' \in Q(t') \\ \wedge \quad \exists t'' \in \mathit{dom}(Q).\mathit{equal}_\eta(t,t'') \wedge \exists \tau'' \in \mathit{compl}(\tau,t'',\eta).\tau'' \notin Q(t'') \end{cases}$$

Definition 1 (Concurrency)**.** *The maximal number of alive candidates at an event is called* concurrency*:*

$$\mathit{concur}_Q(t) = \max_{\eta \in \mathit{eve}(t)} |\{\tau \mid (\tau,\eta) \in \mathit{alive}_Q(t)\}|$$

We say that the concurrency of a query Q is k-bounded (with $k \in \mathbb{N}_0$) if $\mathit{concur}_Q(s) \leq k$ for all structures $s \in \mathit{dom}(Q)$. It is bounded if it is k-bounded for some $k \in \mathbb{N}_0$. Note that queries with unbounded concurrency cannot be processed in streaming manner with bounded memory.

Compared with the original definition by Bar-Yossef et al. (Definition 3 in [BYFJ05]), our notion of concurrency is generalized to n-ary queries, and arbitrary query languages. A consequence is that we deal with partial tuples. We choose to include the empty tuple $\{\bullet\}^n$ among possible alive candidates. The reason is that this simplifies the definitions and complexity analysis, as our algorithms treats the empty tuple as other candidates. By the way, this only introduces a difference of 1 between both definitions, and keeps the bounds unchanged. We note that in this original definition [BYFJ05] resides some ambiguity: It seems that nodes cannot be alive before being closed. From the use of concurrency in the same paper, it appears that the definition of Bar-Yossef et al. has to be interpreted as formally presented above.

For XPath expressions, the concurrency differs from the number of matches. For each alive candidate, there can be numerous matches, i.e. embeddings of the expression into the tree, verifying the axis and label tests of the query. In particular, the concurrency is always lower than the number of matches, as for each match corresponds a unique alive candidate.

3.2.4 Evaluation Model

To formalize our notion of streaming computation of queries, and to have a clean notion of complexity, we define Streaming Random Access Machines (SRAMs), as illustrated in Figure 3.1. These are inspired by RAM machines described by Grandjean et al. [Gra96, GO04].

The purpose of SRAMs is to characterize a class of algorithms for streaming query answering, that we consider as realistic. The lower complexity bounds we will present, apply only to such realistic algorithms. In particular, our model avoids compaction tricks for the storage of nodes, by providing node identifiers only at opening time, and by disabling the access to node identifiers by the finite state control. We detail these features after the following definition. Note that compaction techniques are not used by existing streaming algorithms for general purpose query languages. We assume that the size of every node is in $O(1)$, independently of the length of its address. This is realistic, since we assume trees of bounded depth anyway.

An SRAM is a deterministic machine composed by:

- an *input tape I*, on which the head cannot write nor move to the left,
- an infinite set of *registers* $\{i\}_{i\in\mathbb{N}_0}$. Each register can contain a node. We write $R(i)$ for the content of the register i.
- a *working memory W*, with read/write and constant-time random access
- an *output tape O*, on which the head cannot move to the left nor read
- a finite state control, made of a finite set of instructions. The allowed instructions are:

  all usual instructions of random access machines for reading and writing on the working memory *W*.

  read the event below the head of the input tape *I* is read. Such an event contains three items: an action $\alpha \in \{op, cl\}$, a letter $a \in \Sigma$, and, if $\alpha = op$, a node identifier π. Note that node identifiers may not correspond to our encoding on $\mathbb{N}^*$, and thus the program cannot compute such identifiers.

  - if $\alpha = op$, then the node identifier π is stored in a free register i, i.e. $R(i) \leftarrow \pi$. The data for α, a and i are written on *W*.
  - if $\alpha = cl$, then the data for α and a are written on *W*.

  output if the head of *W* points to $(i_1, \ldots, i_n)$, then $(R(i_1), \ldots, R(i_n))$ is written onto the output tape *O*, and the head of *O* moves to the next free slot.

  free(i) to free the register i, where i is read from *W*.

  halt to stop the machine.

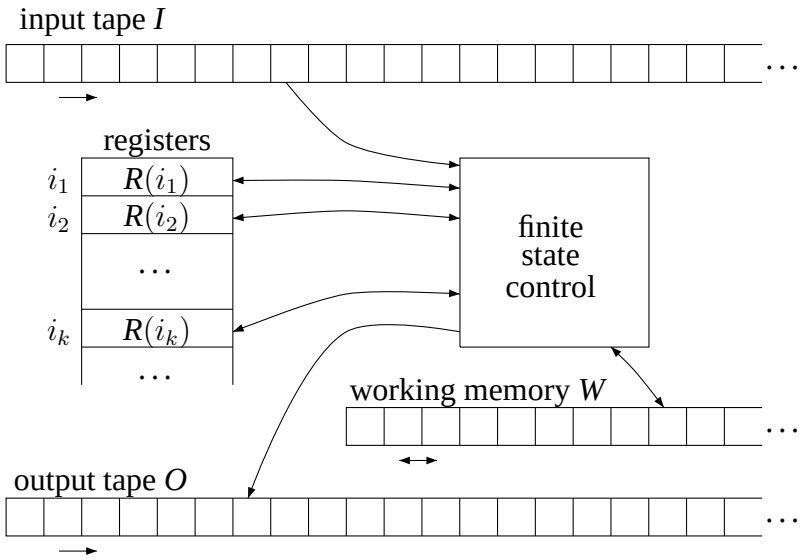


Figure 3.1: Streaming Random Access Machine.

To define the intended inputs, we present a variant of visible words with node identifiers: $\text{vw}_{\text{id}}(t)$ is obtained from $\text{vw}(t)$ by adding the nodes in opening events (we use the symbol $\sharp$ for closing events). This corresponds to the stream generated by the parser, and hence the real input of streaming algorithms. For clarity, we suppose that the parser uses our encoding of nodes as a sequence of integers, i.e. $\mathit{nod}(t)$. For instance for the tree $t = a(b, c)$ we get:

$$\begin{array}{c} \text{XML stream} \quad \texttt{<a>} \quad \texttt{} \quad \texttt{} \quad \texttt{<c>} \quad \texttt{</c>} \quad \texttt{} \\ \text{vw}_{\text{id}}(t) = (\mathit{op}, a, \epsilon)\cdot(\mathit{op}, b, 1)\cdot(\mathit{cl}, b, \sharp)\cdot(\mathit{op}, c, 2)\cdot(\mathit{cl}, c, \sharp)\cdot(\mathit{cl}, a, \sharp) \end{array}$$

Registers are used to capture the number of candidate nodes to be buffered simultaneously. Providing node identifiers only at opening time avoids some hacks in the representation of candidates. Let Q be a monadic query, and assume that Q can determine at closing time whether a node is selected. Then an SRAM $\mathcal{M}$ computing Q can be built, such that $\mathcal{M}$ uses only one register (for the current node), and a stack on the working tape *W* to store candidates, using an internal representation (not node identifiers). Hence the number of registers used by $\mathcal{M}$ do not capture the number of simultaneous candidates. Moreover, the internal representation of candidates on *W* allows compression techniques, that we want to avoid in our model.

Definition 2 (Computation)**.** *An SRAM $\mathcal{M}$* computes *a query Q if for all trees*

$t \in dom(Q)$, if $vw_{id}(t)$ is on the input tape, then $\mathcal{M}$ outputs the set $Q(t)$, in any order but without duplicates, and halts.

Node identifiers cannot be written to the working memory *W*, so they cannot be computed by the finite state control. Even if they could, there would be no way to output them, as only the registers contents can be output. If node identifiers were stored on the working memory, they could be computed with less memory, by tricky methods. Consider for instance the query on words that selects all positions of a word w before a b-position. The concurrency of this query is very high (and even unbounded), as all positions are alive candidate until a b-position is read. However, the query can be computed with memory $O(log(|w|))$, by just maintaining a counter for the current position, and if it is a b, iterate from the last b-position (to be also stored) to the current one. In our model, this trick is impossible, as (identifiers of) alive candidates are stored in registers.

The working memory *W* considers a candidate $(\pi_1, \ldots, \pi_n)$ as the tuple of registers addresses $(i_1, \ldots, i_n)$, with $(\pi_1, \ldots, \pi_n) = (R(i_1), \ldots, R(i_n))$. Note that the set of candidate tuples might be stored in a more compact way (as investigated for instance by Meuss et al. in [MSB01]), but this is usually not the case for algorithms in the literature. This is why we choose to store only nodes in registers, instead of tuples of nodes.

For queries defined by XPath expressions, the implementation by an SRAM does not exactly follow the XPath semantics defined by the W3C. First, the W3C XPath semantics requires that the subtrees rooted at selected nodes should be output, not only node identifiers. Second, the selected (tuples of) nodes should be output in document order. We think however that both requirements are too strong to be integrated inside the machine. The query evaluation algorithms can be used in some transformation language (like XQuery or XSLT), where the materialization of subtrees is not needed at the selection level, and identifiers suffice. Similarly, the document order is not useful in all transformations, and known to be incompatible with efficient stream processing (and for instance with earliest query answering). We choose to keep a model based on the minimal input/output requirements of streaming evaluation of queries. This strengthens our lower bounds and hardness results.

Definition 3 (Complexity)**.** *An SRAM $\mathcal{M}$ computes the query Q with per-event time Time$(\mathcal{M}, t)$ and per-event space Space$(\mathcal{M}, t)$ if $\mathcal{M}$ computes Q, and during the computation of $Q(t)$:*

1. *at all time points, $\mathcal{M}$ uses some registers among registers $R(i)$ with $i \leq$ Space$(\mathcal{M}, t)$, and at most Space$(\mathcal{M}, t)$ slots in the working memory, and*
2. *the number of executed instructions between reading two successive events on the input tape is bounded by Time$(\mathcal{M}, t)$. This includes the time be-*

fore reading the first event, and the time between reading the last one and halting.

Most of the algorithms will have to pass information from opening to closing events. This is usually done through a stack, that has to be stored in the working memory. Hence for these algorithms, the space requirements will be at least the depth of the input tree.

These definitions are done modulo an encoding for the input and output data. The cost of instructions is supposed uniform. The size of each register (i.e. the number of bits that can be stored in a register) is exactly the size of node identifiers in the input stream. As node identifiers cannot be computed by the finite state control, the number of registers that are simultaneously required by monadic queries is at least the concurrency of the query. This gives us a lower bound for memory consumption (Proposition 6 below), thanks to the separation between registers containing node identifiers, and the working memory. A working hypothesis here is that the concurrency is a real lower bound for rich monadic query classes, as proved by Bar-Yossef et al. for an XPath fragment [BYFJ05].

Proposition 6. *Evaluating a monadic query Q on a tree t requires per-event space* $\Omega(\mathit{concur}_Q(t))$.

For n-ary queries, this is not true, as candidate tuples (containing registers identifiers, not nodes) are stored in the working memory. But the registers need to memorize which nodes are used in the alive candidates. We call this quantity $\mathit{concur\_nod}_Q(t)$:

$$\mathit{concur\_nod}_Q(t) = \max_{\eta \in \mathit{eve}(t)} |\{\pi \mid \tau \text{ is alive at } \eta \text{ and } \pi \text{ is in } \tau\}|$$

Proposition 7. *Evaluating a query Q on a tree t requires per-event space* $\Omega(\mathit{concur\_nod}_Q(t))$.

Proof. Let $\mathcal{M}$ be an SRAM computing a query Q. Let $t \in \mathcal{T}_\Sigma$, and consider a candidate tuple $\tau \neq \{\bullet\}^n$ that is alive at event $\eta \in \mathit{eve}(t)$. Let $I^{\preceq \eta}$ be the content of the input tape I before η. Let π be a node appearing in τ, and suppose for contradiction that π is not stored in registers after treating η (i.e., just before reading the event following η). As τ is alive at η, there is a continuation C of the input stream that selects τ. Consider the run of $\mathcal{M}$ on the concatenation of $I^{\preceq \eta}$ and C. As $\mathcal{M}$ is deterministic, the machine is in the same state at η, so π is not in the registers. $\mathcal{M}$ will have to output τ before halting, and τ could not be output before η because it is alive, and thus there is another continuation of the stream for which τ is not selected. Hence τ will have to be output (strictly) after η. The only way to output τ is to use the "output" instruction. But this requires to retrieve

π from the registers. The identifier of π is read only once on the input stream, at event (op, π). As it is not in registers after η, and $(op, \pi) \preceq \eta$ (as π is in τ, alive at η), $\mathcal{M}$ cannot output τ, which contradicts its definition. □

This also proves Proposition 6, because for monadic queries $\mathit{concur\_nod}_Q(t) = \mathit{concur}_Q(t)$.

3.3 Streamable Query Classes

3.3.1 Streamability

We now formally define our notion of streamability, and study some properties of this new notion.

Definition 4 (Streamability)**.** *Let $m \in \mathbb{N}_0 \cup \{\infty\}$. A query class $\mathcal{Q}$ is* m-streamable with polynomials p_0, p_1, p_2 *if one can compute SRAMs $\mathcal{M}(Q_e)$ in time $p_0(|e|)$ for all $e \in \mathcal{Q}$ such that $\mathcal{M}(Q_e)$ computes Q_e and if $\mathit{concur}_{Q_e}(t) \leq m$ then $\mathit{Space}(\mathcal{M}(Q_e), t) \leq p_1(|e|)$ and $\mathit{Time}(\mathcal{M}(Q_e), t) \leq p_2(|e|)$. A query class $\mathcal{Q}$ is* m-streamable *if it is m-streamable for some p_0, p_1, p_2, and* streamable *if it is ∞-streamable.*

We recall that $\mathit{Space}(\mathcal{M}(Q), t)$ and $\mathit{Time}(\mathcal{M}(Q), t)$ are per-event complexity measures. The definition directly provides a hierarchy of streamability for query classes.

Lemma 1. *Let $m \in \mathbb{N}_0$. If the query class $\mathcal{Q}$ is $(m+1)$-streamable then it is also m-streamable, with the same polynomials. Furthermore, if $\mathcal{Q}$ is streamable, then it is m-streamable for all $m \in \mathbb{N}_0$, with the same polynomials.*

Hence we get a hierarchy of query classes:

$$0\text{-streamable} \supseteq 1\text{-streamable} \supseteq \cdots \supseteq m\text{-streamable} \supseteq \cdots \supseteq \text{streamable}$$

However, for classes of monadic queries, streamability may fail even though m-streamability holds for all $m \in \mathbb{N}_0$. Consider for instance the query Q_e defined by the XPath expression $e = /\mathit{self}::a[\mathit{ch}::b]/\mathit{ch}::c$ on trees of $\mathcal{T}_{\{a,b,c,d\}}$. This query selects all c-nodes that are children of an a-labeled root, and have a sibling labeled by b. It is easy to see that Q_e has unbounded concurrency. For instance, on the tree $a(c, \ldots, c, b)$, all c-nodes are alive before opening the b-node. Let $\mathcal{Q} = \{Q_e\}$. This query class is m-streamable for all $m \in \mathbb{N}_0$: For a given m, one can build in PTIME an SRAM $\mathcal{M}(Q_e)$ that uses polynomial per-event space and time, on trees for which the concurrency is less than m. However, $\mathcal{Q}$ is not ∞-streamable. This class is in Star-Free XPath, so by the lower bound of Bar-Yossef

et al. [BYFJ05], any algorithm computing Q_e requires space $\Omega(\mathit{concur}_{Q_e}(t))$ on non-recursive t, i.e., when t does not have a branch with duplicated labels. Hence an SRAM computing Q_e cannot use space bounded by some polynomial p_1, as Q_e has unbounded concurrency. In order to relate streamability and m-streamability, we have to add a condition on the concurrency of the query class.

Definition 5. *A query class $\mathcal{Q}$ has* polynomially bounded concurrency *if there exists a polynomial p such that $\mathit{concur}_{Q_e}(t) \leq p(|e|)$ for all $e \in \mathcal{Q}$ and trees $t \in \mathit{dom}(Q)$.*

Proposition 8 gives a sufficient condition for being streamable: the query class has to be m-streamable for all m with the same polynomials, and must have polynomially bounded concurrency.

Proposition 8. *If the concurrency of a query class $\mathcal{Q}$ is polynomially bounded and there exist polynomials p_0, p_1, p_2 such that for all $m \in \mathbb{N}_0$, $\mathcal{Q}$ is m-streamable with p_0, p_1, p_2, then $\mathcal{Q}$ is ∞-streamable.*

Proof. Let p, p_0, p_1, p_2 be polynomials such that $\mathcal{Q}$ is m-streamable with p_0, p_1, p_2 for all $m \in \mathbb{N}_0$, and the concurrency of $\mathcal{Q}$ is bounded by p. Let $t \in \mathcal{T}_\Sigma$ and $e \in \mathcal{Q}$. For every $m \in \mathbb{N}$, let $\mathcal{M}_m(Q_e)$ be an SRAM computing Q_e and verifying m-streamability. We show that $\mathcal{M}_{p(|e|)}(Q_e)$ verifies ∞-streaming with polynomials p'_0, p_1, p_2 where $p'_0(X) = p_0(X)+X+|p|$. To generate the SRAM $\mathcal{M}_{p(|e|)}(Q_e)$ in time $p_0(|e|)+|e|+|p|$, we first compute the value of $p(|e|)$ in time $|e| + |p|$, and then generate $\mathcal{M}_{p(|e|)}(Q_e)$ in time $p_0(|e|)$. A single step of $\mathcal{M}_{p(|e|)}(Q_e)$ on trees $t \in \mathcal{T}_\Sigma$ costs $\mathit{Space}(\mathcal{M}_{p(|e|)}(Q_e), t) \leq p_1(|e|)$ and $\mathit{Time}(\mathcal{M}_{p(|e|)}(Q_e), t) \leq p_2(|e|)$, as bounded concurrency yields $\mathit{concur}_{Q_e}(t) \leq p(|e|)$. □

For the converse, we have already seen in Lemma 1 that ∞-streamability implies m-streamability for all $m \in \mathbb{N}_0$, with the same polynomials. We can also prove a weaker form of bounded concurrency.

Proposition 9. *If a query class $\mathcal{Q}$ is ∞-streamable, there exists a polynomial p such that for all $e \in \mathcal{Q}$ and all $t \in \mathit{dom}(Q_e)$, $\mathit{concur\_nod}_{Q_e}(t) \leq p(|e|)$.*

Proof. Suppose that $\mathcal{Q}$ is ∞-streamable, and let p_1 be the corresponding polynomial bounding space. By Proposition 7 we get, for all $e \in \mathcal{Q}$ and $t \in \mathit{dom}(Q_e)$, and all for SRAMs $\mathcal{M}(Q_e)$ computing Q_e:

$$\mathit{concur\_nod}_{Q_e}(t) \leq \mathit{Space}(\mathcal{M}(Q_e), t) \leq p_1(|e|)$$

so that *concur_nod* is polynomially bounded by p_1. □

Benedikt and Jeffrey [BJ07] exhibited two ∞-streamable query classes. Both are fragments of backward $\mathcal{X}_{\text{until}}$, an extension of CoreXPath 1.0 adding an *until* operator, but restricted to backward and downward axes. The authors prove the ∞-streamability of two query classes:

1. Boolean queries (i.e. filters) defined by backward $\mathcal{X}_{\text{until}}$ formulas, over non-recursive trees, and

2. monadic queries defined by strict backward $\mathcal{X}_{\text{until}}$ formulas, over non-recursive trees. *Strict* means that downward axes are not allowed. It implies that concurrency is at most one, as all conditions to be satisfied for selecting a new candidate depend on the prefix until this candidate (no look-ahead is needed).

3.3.2 Boolean and Monadic Queries

For Boolean and monadic queries, some properties of the concurrency give stronger results. Boolean queries have a concurrency bounded by 1, as the only possible alive candidate is the empty tuple $()$ (which can be seen as either the potentially selected tuple, or the empty partial candidate $\{\bullet\}^0$).

Proposition 10. *A Boolean query class $\mathcal{Q}$ is streamable if and only if $\mathcal{Q}$ is 1-streamable.*

Proof. Suppose that $\mathcal{Q}$ is 1-streamable. Then it is 0-streamable by Lemma 1. As the concurrency of Boolean queries is bounded by 1, $\mathcal{Q}$ is m-streamable for all $m \in \mathbb{N}_0$, with the same polynomials, and by Proposition 8, it is ∞-streamable. The converse is immediate by Lemma 1. □

For monadic queries, the concurrency may be unbounded in the general case. However, both forms of concurrency we introduced coincide, and we get the following equivalence.

Corollary 1. *A monadic query class $\mathcal{Q}$ is streamable if and only if the concurrency of $\mathcal{Q}$ is polynomially bounded and there exist polynomials p_0, p_1, p_2 such that for all $m \in \mathbb{N}_0$, $\mathcal{Q}$ is m-streamable with p_0, p_1, p_2.*

Proof. Immediate by Proposition 8, Lemma 1, Proposition 9, and the fact that $\mathit{concur\_nod}_Q(t) = \mathit{concur}_Q(t)$ for monadic queries. □

3.4 Hardness of Streamability

We present hardness results for streamability of small classes of queries. Of course these hardness results also hold for larger query classes. We start by studying the complexity of deciding whether a query class has bounded concurrency (resp. polynomially bounded concurrency). We then investigate the streamability of queries defined by XPath expressions. In this section we only consider queries Q with universal schema $\mathit{dom}(Q) = \mathcal{T}_\Sigma$, and the results also hold for queries with other schemas.

3.4.1 Hardness of Bounded Concurrency

We start by defining, from any set of monadic queries, another set of monadic queries that requires high buffering. The idea is to start from a monadic query Q, and define the query $\mathit{all}(Q)$ that selects all the children of the root, if Q has a match when evaluated from the last child of the root.

We call a monadic query Q *descending* if node selection by Q is independent of the node's upper context, i.e. if $\pi \in Q(t)$ is equivalent to $\epsilon \in Q(t.\pi)$, where $t.\pi$ is the subtree of t rooted at π. For all monadic queries Q we define another monadic query $\mathit{all}(Q)$ whose semantics is given by the following XPath expression:

$$\mathit{all}(Q) =_{\mathit{df}} /\mathit{ch}::*[\mathit{ns}^*::*[Q][\mathit{not}\,(\mathit{ns}::*)]]$$

It selects all children of the root if the last child of the root belongs to the language of the Boolean query $[Q]$, which is $L_{[Q]} = \{t \in \mathcal{T}_\Sigma \mid Q(t) \neq \emptyset\}$. Let $\mathcal{Q}$ be a language of monadic queries. We say that the operator *all* can be defined polynomially in $\mathcal{Q}$ if there exists a polynomial p such that for all $e \in \mathcal{Q}$ there exists an expression $e' \in \mathcal{Q}$ of size at most $p(|e|)$ such that $Q_{e'} = \mathit{all}(Q_e)$. We say that a node π is safely selected (resp. safely rejected) by a query at event η if π is selected (resp. not selected) in all valid continuations of the stream after η.

Lemma 2. *For all descending monadic queries Q, trees t matching $a(t_1, \ldots, t_j)$, and $1 \leq k \leq j$:*

1. *node k is safely selected by $\mathit{all}(Q)$ at (op, j) in t iff $L_{[Q]} = \mathcal{T}_\Sigma$.*
2. *node k is safely rejected by $\mathit{all}(Q)$ at (op, j) in t iff $L_{[Q]} = \emptyset$.*
3. *node k is alive for $\mathit{all}(Q)$ at (op, j) in t iff $\emptyset \neq L_{[Q]} \neq \mathcal{T}_\Sigma$.*

Proof. ($\Leftarrow$) We first assume that $L_{[Q]} = \mathcal{T}_\Sigma$ and show that k is safely selected at event (op, j) in trees t matching $a(t_1, \ldots, t_j)$ and $1 \leq k \leq j$. Let $t' \in \mathcal{T}_\Sigma$ be a continuation of t beyond (op, j), i.e. $\mathit{equal}_{(\mathit{op},j)}(t, t')$ holds. Let j' be the last child

of the root of t', so that $j \leq j'$. Then $k \in \mathit{all}(Q)(t')$ iff $j' \in [\![[Q]]\!]^{t'}_{\text{filter}}$. The latter is equivalent to $\epsilon \in [\![[Q]]\!]^{t'.j'}_{\text{filter}}$ since Q is descending. This holds since $L_{[Q]} = \mathcal{T}_\Sigma$. Thus event (op, j) is sufficient for selection of k in all continuations of t.

We now assume that $L_{[Q]} = \emptyset$. The last child of the root cannot satisfy $[Q]$ in any continuation, so no node can ever be selected.

We suppose that $\emptyset \neq L_{[Q]} \neq \mathcal{T}_\Sigma$ and show that k is alive at (op, j) in trees $t \in \mathcal{T}_\Sigma$ with $j \in \mathit{nod}(t)$ and $1 \leq k \leq j$. Let $t' \in \mathcal{T}_\Sigma$ be a continuation of t beyond (op, j) and j' be the last child of t'. Now k is selected if and only if $t'.j' \in L_{[Q]}$. Since $\emptyset \neq L_{[Q]} \neq \mathcal{T}_\Sigma$ this is the case in some t' but not in others, so that k is alive at (op, j).

($\Rightarrow$) Since these cases are exhaustive, all inverse implications follow. □

As a consequence, the concurrency of the query defined by $\mathit{all}(Q)$ is bounded only if $L_{[Q]}$ is empty or universal, as for $t = a(t_1, \ldots, t_j)$ we get:

$$\mathit{concur}_{\mathit{all}(Q)}(t) = \begin{cases} 0 & \text{if } L_{[Q]} = \emptyset \\ 1 & \text{if } L_{[Q]} = \mathcal{T}_\Sigma \\ j+1 & \text{otherwise} \end{cases}$$

The concurrency is 1 when $L_{[Q]} = \mathcal{T}_\Sigma$, because in this case the empty candidate $(\bullet)$ is always alive. It is never alive on an empty query, i.e., whenever $L_{[Q]} = \emptyset$.

Proposition 11. *Let $\mathcal{Q}$ be a class of descending monadic queries that can define operators "all" and "not" in polynomial time, then the two decision problems below are more difficult modulo a* PTIME *reduction than universality $L_{Q_{[e]}} = \mathcal{T}_\Sigma$ for all $e \in \mathcal{Q}$.*

Polynomially bounded concurrency

PARAMETER:	$\mathcal{Q}$
INPUT:	$e \in \mathcal{Q}$
OUTPUT:	*decide whether there exists a polynomial p such that* : $\forall t \in \mathcal{T}_\Sigma.\ \mathit{concur}_{Q_e}(t) < p(\lvert e \rvert)$

Bounded concurrency

PARAMETER:	$\mathcal{Q}$
INPUT:	$e \in \mathcal{Q}$
OUTPUT:	*truth value of:* $\exists k \in \mathbb{N}_0.\ \forall t \in \mathcal{T}_\Sigma.\ \mathit{concur}_{Q_e}(t) < k$

Proof. Since all queries defined by $e \in \mathcal{Q}$ are descending, the existence of a polynomial p such that $\forall t.\, \mathit{concur}_{\mathit{all}(Q_e)}(t) \leq p(|e|)$ is equivalent to $L_{Q_{[e]}} = \mathcal{T}_\Sigma \vee L_{[not(e)]} = \mathcal{T}_\Sigma$ by Lemma 2; equally for $\exists k \in \mathbb{N}_0.\ \forall t \in \mathcal{T}_\Sigma.\ \mathit{concur}_{\mathit{all}(Q_e)}(t) < k$. □

Proposition 11 gives a first result on the hardness of deciding bounded concurrency of queries. For deterministic automata, testing the universality is in PTIME, and we will see in Chapter 7 that deciding bounded concurrency is also in PTIME.

3.4.2 Hardness of Streamability

We now characterize the streamability of query classes $\mathcal{Q}$. The following theorem states that being 1-streamable (while verifying two other properties) implies that the universality of descending Boolean queries defined from $\mathcal{Q}$ is in PTIME. This can be used to prove that some query class is not 1-streamable, and hence not m-streamable for any $m \in \mathbb{N} \cup \{\infty\}$.

Theorem 1. *Let $\mathcal{Q}$ be a class of definitions of monadic queries such that there exist polynomials r, s such that:*

1. *query $\mathit{all}(Q_e)$ is definable by an expression in $\mathcal{Q}$ of size $r(|e|)$ in time $O(r(|e|))$.*
2. *membership $a \in L_{Q_{[e]}}$ can be tested in time $s(|e|)$ for all $a \in \Sigma$.*

If such a class $\mathcal{Q}$ is 1-streamable with polynomials p_0, p_1, p_2 then the universality problem of Boolean queries $\{Q_{[e]} \mid e \in \mathcal{Q} \text{ descending}\}$ can be solved in polynomial time $O(p_0(r(|e|)) + s(|e|) + r(|e|) + p_1(r(|e|)) \cdot p_2(r(|e|)))$.

Proof. Our polynomial time equivalence test for descending queries defined in $\mathcal{Q}$ works as follows:

```
fun univ_Q(e)            # where e ∈ Q descending
  let a∈Σ arbitrary
  if a in L_{Q_[e]}
  then                   # language non-empty
    compute e' with Q_{e'} = all(Q_e)
    let j = p1(|e'|)+1
    let t=a(a,...,a) with j children
    let M = M(Q_{e'})    # needs time p0(|e'|)
    let out = run M on t until event (op,j)
      if out.isEmpty()
      then return false
      else return true
  else                   # language non-universal
    return false
```

Testing whether a belongs to $L_{Q_{[e]}}$ can be done in time $s(|e|)$. The construction of e' defining $\mathit{all}(Q_e)$ with size $|e'| = r(|e|)$ requires time $O(r(|e|))$. The whole algorithm requires time $O(p_0(r(|e|)) + s(|e|) + r(|e|) + j \cdot p_2(|e'|))$, which is $O(p_0(r(|e|)) + s(|e|) + r(|e|) + p_1(r(|e|)) \cdot p_2(r(|e|)))$. It remains to argue the correctness of the algorithm.

Case $L_{Q_{[e]}} = \mathcal{T}_\Sigma$. Since e is descending, we have $\mathit{concur}_{Q_{e'}}(t) = 1$ for $t = a(a, \ldots, a)$ with j children from Lemma 2. Since $\mathcal{Q}$ is streamable modulo 1-concurrency, there exists an SRAM $\mathcal{M}(Q_{e'})$ that requires on input trees t space at most $p_1(|e'|)$ and time at most $p_2(|e'|)$ per step. All nodes $k \in \mathit{nod}(t)$ where $1 \leq k \leq j$ are safely selected by $Q_{e'} = \mathit{all}(Q_e)$ at event

(op, j) by part 1 of Lemma 2. These are $p_1(|e'|) + 1$ many nodes, but the space of $\mathcal{M}(Q_{e'})$ is at most $j = p_1(|e'|) + 1$. Since none of the nodes can be discarded, one of them must be output until (op, j). Thus $out \neq \emptyset$ and our algorithm returns *true* as expected.

Case $L_{Q_{[e]}} \neq \mathcal{T}_\Sigma$. If $a \notin L_{Q_{[e]}}$ then we know that $L_{Q_{[e]}}$ is not universal and can safely return *false*. Otherwise, SRAM $\mathcal{M}(Q_{e'})$ is run on t, but cannot output anything until event (op, j) since all nodes $k \in \mathit{nod}(t)$ with $1 \leq k \leq j$ are still alive for $Q_{e'} = \mathit{all}(Q_e)$ by part 2 of Lemma 2. Thus, $out.isEmpty()$ is true so that our algorithm returns *false* as expected.

□

3.4.3 Non-Streamability of Forward XPath

We now apply the previous results on Forward XPath. First, this proves that bounded concurrency and polynomially bounded concurrency can not be decided in PTIME, unless PTIME = NP.

Corollary 2. *Deciding bounded concurrency resp. polynomially bounded concurrency is coNP-hard for monadic queries in Forward XPath.*

Proof. Universality for a fragment of Forward XPath (using only downward axes) is coNP-hard by Proposition 4. So the corollary follows from Proposition 11. □

In terms of streamability, we also get a negative result for Forward XPath.

Corollary 3. *Forward XPath is* not 1*-streamable except if P=NP.*

Proof. Forward XPath permits to define the operator *all* in linear time. Universality of $[e]$ is equivalent to universality of $/\mathit{ch}::*[e]$, which is descending for all Forward XPath queries e. The universality problem for monadic descending Forward XPath queries in the fragment is coNP-hard (by Proposition 4). Theorem 1 thus shows that this query class is not 1-streamable except if PTIME = NP. □

This shows that even the weak notion of 1-streamability is unfeasible for Forward XPath. In Chapter 6, we define fragments of Forward XPath that are m-streamable for all $m \in \mathbb{N}_0$.

3.5 Conclusion

In this chapter we defined our computational model for query answering through a special form of RAMs called SRAMs. Based on this model and the notion of concurrency of queries, we introduce a measure of streamability for classes of queries. This classifies query classes in the following way. For query classes that are not 0-streamable, there is no PTIME algorithm detecting empty queries, and thus memory consumption cannot be optimal with PTIME processing. Query classes that are m-streamable with $m \in \mathbb{N}_0$ allow a polynomial space and time evaluation for queries with concurrency at most m. ∞-streamable queries enjoy this property for all queries of the class. We can observe that the definition varies from coarse-grained static requirements for ∞-streamability to more fine-grained requirements for m-streamability, where the algorithm is supposed to evaluate queries efficiently only on trees implying low concurrency.

The study of necessary and sufficient conditions for ∞-streamability reveals some asymmetry between monadic and n-ary queries. For n-ary queries, we have to distinguish between $\mathit{concur}_Q(t)$, the number of simultaneous alive tuples, and $\mathit{concur\_nod}_Q(t)$, the number of nodes involved simultaneously in alive tuples. This comes from the definition of SRAMs, where registers do not store candidates (i.e. tuples) but node identifiers used by candidates. The reason of this design choice is that in real algorithms, tuples might be represented compactly, and in general the concurrency is not a lower bound for evaluating queries. Concurrency is proved to be a lower bound only on some fragments of XPath [BYFJ05]. An interesting question would be to prove that $\mathit{concur\_nod}_Q(t)$ is a lower bound for large classes of n-ary queries, which we conjecture to be true for large query classes. For this, we would have to find fooling sets in order to apply results from communication complexity.

We have seen at the end of this chapter some negative results about Forward XPath: it is coNP-hard to decide the bounded concurrency for monadic queries, and Forward XPath is not 1-streamable. In Chapter 6, we define fragments that are m-streamable for all $m \in \mathbb{N}_0$, and another ∞-streamable fragment. In Chapter 5, we study the streamability of queries defined by Streaming Tree Automata.

Chapter 4

Streaming Tree Automata

Contents

4.1 Introduction

Tree automata are acceptors for trees over a given alphabet. While being procedural objects, they enjoy clean relations with logics and language theory [CDG$^+$07]. Hence they can be considered either for algorithms (they are based on notions of runs) or for specification (they define tree languages).

In this manuscript, we will use both aspects of tree automata. In particular, tree automata will define queries, and will also serve as basis for our algorithms. For this reason, we are looking for tree automata whose runs can operate

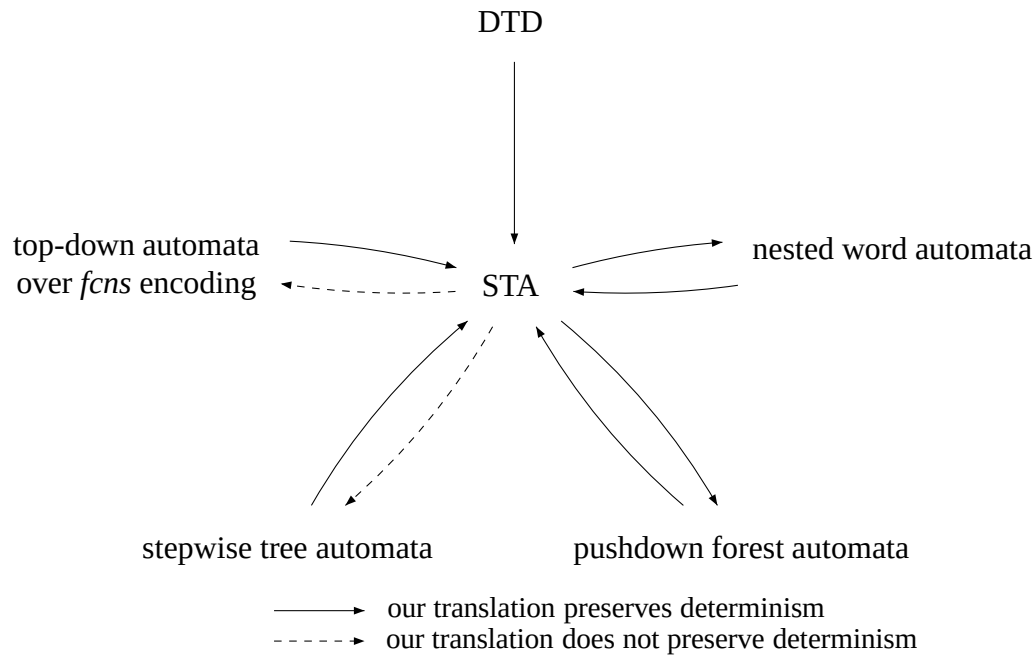


Figure 4.1: Translations provided in this chapter.

on XML streams, and thus respect a pre-order traversal of trees. Tree automata usually operate bottom-up (from the leaves to the root of the tree) or top-down. Some automata models operating in pre-order were however proposed for tree-like structures. Neumann and Seidl propose *pushdown forest automata* (PFAs) [NS98], a notion of automata for hedges, which generalize unranked trees. These automata were sometimes adapted to particular algorithmic contexts: They are reformulated to Pre-Order Automata by Berlea in [Ber06], and to Non-Uniform Automata by Frisch in [Fri04]. More recently, Alur and Madhusudan introduce *visibly pushdown automata* (VPAs) [AM04] in the context of program verification. This model is also used for XML streams processing [KMV07]. VPAs were reformulated to *nested word automata* (NWAs) by Alur [Alu07]. All these models do not operate directly on trees. PFAs operate on hedges, VPAs on words over a visible alphabet (where each letter either always pushes or always pops data onto the stack), and NWAs on nested words, i.e. words with a binary nesting relation on positions.

In this chapter, we define *Streaming Tree Automata* (STAs), a notion of automata operating directly on unranked trees in pre-order. STAs are a reformulation of NWAs, that operate directly on trees, instead of nested words. We start by showing how DTDs can be translated to STAs. We then relate them to PFAs and NWAs by providing the back and forth translations towards these models. We

also study the relationship between STAs and tree automata models that does not operate in streaming order. We provide back and forth translations for two such models. The first one is stepwise tree automata [CNT04], which correspond to standard bottom-up automata on the Curryfication of trees. The second one is the notion of top-down automata on the first-child next-sibling encoding of trees. We show in particular that the translations from both models to STAs preserve determinism, and hence that determinism of STAs is a stronger notion than for these two models. In [AM09], Alur and Madhusudan claim that a stepwise tree automaton can be translated into a NWA with the same number of states, but without providing the translation. The translations provided in this chapter are illustrated in Figure 4.1.

Thanks to these explicit translations, we fix the precise relations between automata notions, as for instance between NWAs and PFAs. Our translations permit to reuse algorithms designed for a specific automata notion with other automata. For instance, queries defined by NWAs can be processed by query answering algorithms for PFAs [BS04].

Throughout this manuscript, we will show the relevance of STAs for stream processing of XML documents. In particular, deterministic STAs define queries that enjoy remarkable streamability properties. In Chapter 5, we propose an efficient query answering algorithm for queries by dSTAs, and prove the m-streamability of this query class for all $m \in \mathbb{N}_0$, on shallow trees. In Chapter 6, we define fragments of XPath, and prove their streamability by translation to dSTAs. In this translation, STAs are able to deterministically detect ends of scopes (regions of trees where matches of XPath expressions can occur). Finally, in Chapter 7, STAs are used to recognize some relations on trees, that we need to prove decidability results. For instance, testing the equality of two tree prefixes until an event is performed by a simple dSTA.

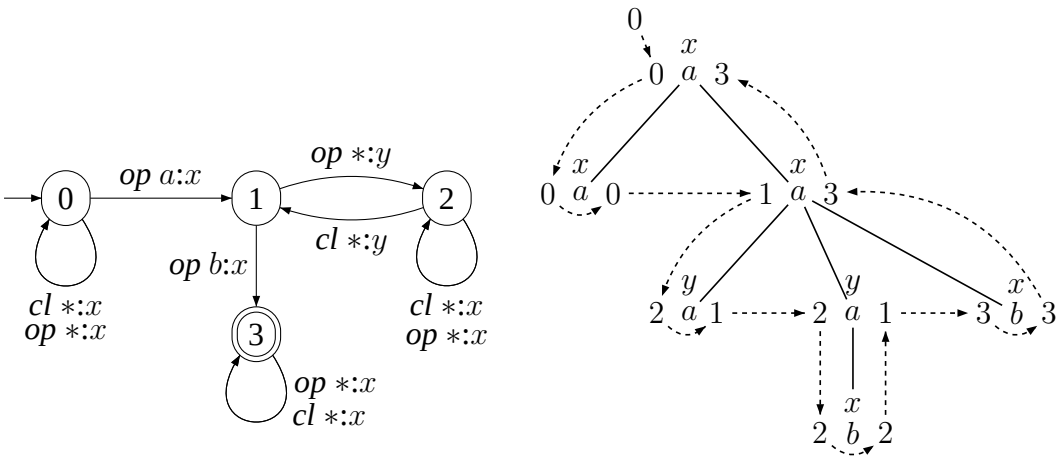
4.2 Streaming Tree Automata

4.2.1 Definition

We begin this chapter with the definition of Streaming Tree Automata (STAs) and their corresponding notion of run.

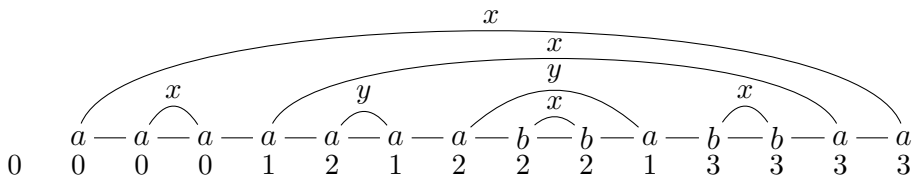
Definition 6. *An STA* $A = (\Sigma, \mathit{stat}, \mathit{init}, \mathit{fin}, \mathit{rul})$ *consists of:*

- *a finite alphabet* Σ *of node labels,*
- *a finite set* $\mathit{stat} = \mathit{stat}_e \uplus \mathit{stat}_n$ *composed of event states* stat_e *and node states* stat_n*,*



(a) An STA A.

(b) Successful run of A on a tree t.



(c) Successful run of A on the nested word of t.

Figure 4.2: An STA checking the Boolean XPath filter $[ch^*::a[ch::b]]$.

- *initial states* $init \subseteq stat_e$ *and final states* $fin \subseteq stat_e$,
- *a set* $rul \subseteq \{op, cl\} \times \Sigma \times stat_n \times stat_e^2$ *of rules. We denote rules as:*

$$q_0 \xrightarrow{\alpha\ a:\gamma} q_1$$

where $\alpha \in \{op, cl\}$, $q_0, q_1 \in stat_e$, $a \in \Sigma$, $\gamma \in stat_n$.

Whenever necessary, we will upper index components of A, as for instance, writing rul^A instead of rul. The size of an STA is its number of rules and states: $|A| = |rul^A| + |stat^A|$. An STA traverses the sequence of events of a given tree t, while annotating all events of t by event states and all nodes of t by node states. Let q_0 be the state of the previous event processed, and (α, π) be the current event. The automaton chooses some rule with action α and label $a = lab^t(\pi)$ whose left hand side is q_0. If $\alpha = op$ then it annotates the node π with node state γ. If $\alpha = cl$ then the rule matches only, if the node state annotated at opening time to π is equal to the node state γ of the rule. For matching rules, the automaton annotates state q_1 on the right hand side to the current event.

Runs More formally, a run r of an STA on a tree t is a pair of functions $(r_{\mathrm{e}}, r_{\mathrm{n}})$ with types $r_{\mathrm{e}} : \mathit{eve}(t) \to \mathit{stat}_{\mathrm{e}}$ and $r_{\mathrm{n}} : \mathit{nod}(t) \to \mathit{stat}_{\mathrm{n}}$, such that $r_{\mathrm{e}}(\mathit{start}) \in \mathit{init}$ and the following rules belong to rul for all $\pi \in \mathit{nod}(t)$ with $a = \mathit{lab}^t(\pi)$, and actions $\alpha \in \{\mathit{op}, \mathit{cl}\}$:

$$r_{\mathrm{e}}(\mathit{pr}((\alpha, \pi))) \xrightarrow{\alpha\ a:r_{\mathrm{n}}(\pi)} r_{\mathrm{e}}((\alpha, \pi))$$

where *pr* returns the preceding event. An example of a run of an STA on the tree $a(a, a(a, a(b), b))$ is given in Figure 4.2. It tests whether this tree satisfies the Boolean XPath query $[\mathit{ch}^*::a[\mathit{ch}::b]]$, or equivalently the first-order formula $\exists x.\ (\mathit{lab}_a(x) \wedge \exists y.\ (\mathit{ch}(x, y) \wedge \mathit{lab}_b(y)))$. When opening an a-node in its initial state 0, this STA guesses whether it matches the a-position of the XPath expression (state 1) or not (state 0). From state 1, it waits while traversing a sequence of states $(2^*1)^*$, until some b-child is opened, before concluding success in state 3. The information of being a child of the a-node opened in state 1 is annotated by node state y, and passed over from the left to the right.

A run r of A on a tree t is successful if $r_{\mathrm{e}}((\mathit{cl}, \epsilon)) \in \mathit{fin}^A$. The set of all possible runs of the STA A on the tree t is denoted $\mathit{runs}^A(t)$ and the subset of all successful runs by $\mathit{runs\_succ}^A(t)$. The recognized language $L(A)$ is the set of all trees $t \in \mathcal{T}_\Sigma$ that permit a successful run by A, i.e., $L(A) = \{t \in \mathcal{T}_\Sigma \mid \mathit{runs\_succ}^A(t) \neq \emptyset\}$. For a hedge $(t_1, \ldots, t_k)$, a run is successful if $r_{\mathrm{e}}(\mathit{start}) \in \mathit{init}^A$ and $r_{\mathrm{e}}((\mathit{cl}, k)) \in \mathit{fin}^A$.

Determinism An STA is *deterministic* or a *dSTA* , if it has a single initial state, no two *op* rules for the same letter use the same event state on the left, and no two *cl* rules for the same letter use the same node state and the same event state on the left. Every STA has an equivalent dSTA, as proved in Section 4.2.2.

Run Computation and Stack The unique run of a dSTA A on a tree t can be computed in a streaming manner, if it exists. The input is the ordered set of events $\mathit{eve}(t)$ for some t obtained by parallel preprocessing with a SAX parser, and the output is the sequence of states that A assigns to the events of t. The comparison between the run of a dSTA on events and on the corresponding nested word is illustrated in Figure 4.2(c). We study the link between STAs and nested word automata in more details, in Section 4.4. The common way to implement an STA is to use a current event state and a stack, in order to store the node states associated to ancestors of the current node, as these states will be used when closing these ancestors. In SRAMs, this stack will be stored inside the working tape.

Weakness Following [Alu07], we call an STA *weak* if $\mathit{stat}_{\mathrm{n}} = \mathit{stat}_{\mathrm{e}}$ and all *op*-rules have the form $q_0 \xrightarrow{\mathit{op}\ a:q_0} q_1$. As proved in Theorem 1 of [Alu07] for NWAs,

every STA A is equivalent to some weak STA B. For instance we can build B of size at most $|B| = O(|\mathit{stat}^A_{\mathrm{e}}| \cdot |\mathit{stat}^A_{\mathrm{n}}|)$. To see this, let $\mathit{stat}^B_{\mathrm{n}} = \mathit{stat}^B_{\mathrm{e}} = \mathit{stat}^A_{\mathrm{e}} \times \mathit{stat}^A_{\mathrm{n}}$, with $\mathit{init}^B = \mathit{init}^A \times \mathit{stat}^A_{\mathrm{n}}$ and $\mathit{fin}^B = \mathit{fin}^A \times \mathit{stat}^A_{\mathrm{n}}$. The rules of B are derived from those of A according to the following two inference schemas.

$$\frac{q_0 \xrightarrow{\mathit{op}\ a:\gamma_1} q_1 \in \mathit{rul}^A \quad \gamma_2 \in \mathit{stat}^A_{\mathrm{n}}}{(q_0,\gamma_1) \xrightarrow{\mathit{op}\ a:(q_0,\gamma_1)} (q_1,\gamma_2) \in \mathit{rul}^B} \qquad \frac{\begin{array}{c} q_0 \xrightarrow{\mathit{cl}\ a:\gamma_0} q_1 \in \mathit{rul}^A \\ \gamma_1,\gamma_2 \in \mathit{stat}^A_{\mathrm{n}} \qquad q_2 \in \mathit{stat}^A_{\mathrm{e}} \end{array}}{(q_0,\gamma_1) \xrightarrow{\mathit{cl}\ a:(q_2,\gamma_0)} (q_1,\gamma_2) \in \mathit{rul}^B}$$

4.2.2 Determinization

We present here the determinization of STAs inspired from the determinization of VPAs [AM04]. This procedure is slightly simpler because we only consider (encodings of) trees, and choose a more algebraic construction. Hence the states of the dSTA will reflect the accessibility relation through the hedge of left siblings. The accessibility relation of an STA A through a hedge $h \in \mathcal{H}_\Sigma$ is the set of pairs $(q_1,q_2) \in \mathit{stat}^A \times \mathit{stat}^A$ such that there is a run of A through h that begins in q_1 and ends in q_2.

Proposition 12. *For every STA A, a dSTA A' recognizing the same language can be computed in time $O(2^{|A|^2})$.*

Proof. A state of A' is a set of pairs of states: $\mathit{stat}^{A'} = 2^{\mathit{stat}^A \times \mathit{stat}^A}$. For such a state $P \in \mathit{stat}^{A'}$, we write $\Pi_1(P) = \{q \mid \exists q'.\ (q,q') \in P\}$ (same for Π_2). In the following, $\mathit{id}_{\mathit{stat}^A}$ denotes $\{(p,p) \mid p \in \mathit{stat}^A\}$, and similarly for $\mathit{id}_{\mathit{init}^A}$. For every state $P \in \mathit{stat}^{A'}$ and label $a \in \Sigma$, we also define Update^a_P by:

$$\mathit{Update}^a_P = \{(q,q') \mid \exists (q_1,q_2) \in P.\ \exists \gamma.\ q \xrightarrow{\mathit{op}\ a:\gamma} q_1 \in \mathit{rul}^A \wedge q_2 \xrightarrow{\mathit{cl}\ a:\gamma} q' \in \mathit{rul}^A\}$$

In other words, if P is the set of pairs of states (q_1,q_2) such that there is a run of A from q_1 to q_2 through the hedge $(t_1,\dots,t_k)$, then Update^a_P is the set of pairs of states (q'_1,q'_2) for which there is a run of A from q'_1 to q'_2 through the tree $a(t_1,\dots,t_k)$, as illustrated in Figure 4.3. We define A' by:

$$\begin{aligned} \mathit{init}^{A'} &= \mathit{id}_{\mathit{init}^A} \\ \mathit{fin}^{A'} &= \{P \mid \pi_2(P) \cap \mathit{fin}^A \neq \emptyset\} \end{aligned}$$

$$\frac{a \in \Sigma \qquad P \in \mathit{stat}^{A'}}{P \xrightarrow{\mathit{op}\ a:P} \mathit{id}_{\mathit{stat}^A} \in \mathit{rul}^{A'}} \qquad \frac{a \in \Sigma \qquad P, P' \subseteq \mathit{stat}^A}{P \xrightarrow{\mathit{cl}\ a:P'} \mathit{Update}^a_P \circ P' \in \mathit{rul}^{A'}}$$

A' is deterministic, and weak. For every $\eta = (\alpha,\pi)$, we write h_η for the hedge whose roots are left siblings of π (including π iff $\alpha = \mathit{cl}$). We prove that the following property is an invariant. From the definition of initial and final states of A', this is sufficient to prove the correctness of the construction.

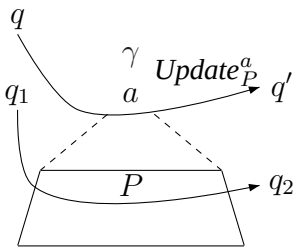


Figure 4.3: Update^a_P.

Invariant: for $r = (r_e, r_n)$ run of A' on t, and $\pi \in \mathit{nod}(t)$:

$$\begin{aligned} r_n(\pi) &= \text{accessibility relation through } h_{(\mathit{op},\pi)} \text{ and} \\ r_e((\mathit{cl},\pi)) &= \text{accessibility relation through } h_{(\mathit{cl},\pi)} \end{aligned}$$

At opening of the root, the state is the identity of initial states, which corresponds to accessibility through an empty hedge at the root.

Suppose that the property holds for events preceding $\eta \in \mathit{eve}(t)$, and that $\eta = (\mathit{op}, \pi)$. If $\mathit{pr}(\eta) = (\mathit{op}, \pi')$ then π is a first child and $r_n(\pi) = r_e((\mathit{op}, \pi')) = id_{\mathit{stat}^A}$, which is the accessibility relation through the empty hedge h_η. Otherwise, if $\mathit{pr}(\eta) = (\mathit{cl}, \pi')$, then by induction hypothesis $r_n(\pi) = r_e((\mathit{cl}, \pi'))$ is the accessibility relation through the hedge $h_{(\mathit{cl},\pi')} = h_\eta$.

Now suppose that $\eta = (\mathit{cl}, \pi)$ and $\mathit{lab}^t(\pi) = a$. Let $\eta' = \mathit{pr}(\eta)$ and $P = r_e(\eta')$. By induction hypothesis, $r_n(\pi)$ is the accessibility through $h_{(\mathit{op},\pi)}$, so it only remains to show that Update^a_P is the accessibility through the hedge $(t.\pi)$ where $t.\pi$ is the subtree of t rooted at π. If π is a leaf then $P = id_{\mathit{stat}^A}$. and Update^a_P is the accessibility through the hedge (a). If π is not a leaf, then by induction hypothesis, P is the accessibility through the hedge of children of π, so Update^a_P is the accessibility through (t_π). □

Note that this procedure is close to optimal, in the sense that there exists a family of regular tree languages L_s (for $s \geq 1$) such that L_s can be recognized by an STA of size $O(s)$, but every dSTA recognizing L_s requires at least 2^{s^2} states [AM09].

4.2.3 Expressiveness and Decision Problems

In terms of expressiveness, STAs capture all MSO-definable tree languages.

Proposition 13. *STAs and MSO capture the same class of languages of unranked trees.*

The logical operations can be performed with the same complexity as for usual tree automata.

Proposition 14. *Union and intersection of STAs can be performed in* PTIME. *Complementation of STAs is* EXPTIME*-complete, and in* PTIME *for dSTAs.*

The complexity of inclusion and universality for STAs is EXPTIME-complete, as other common automata models over unranked trees.

Proposition 15. *Universality and inclusion are both* EXPTIME*-complete problems for STAs, and are in* PTIME *for dSTAs.*

All these results will be proved by the PTIME back and forth translations between STAs and other automata models (stepwise tree automata, for instance) provided in the sequel.

4.3 Translation of DTDs into STAs

In our algorithms, we often consider that schemas are provided by deterministic STAs. They can be obtained by translating extended DTDs that are restrained competition and deterministic [KMV07], so that running such STAs performs one-pass typing. We present the translation of DTDs to STAs. Given a deterministic DTD with alphabet Σ, we compute the collection of Glushkov automata $(G_a)_{a\in\Sigma}$ over Σ, which are deterministic finite automata for the regular expressions of the DTD [BK93]. Let *root* $\in \Sigma$ be the root symbol of the DTD.

From the collection of Glushkov automata, we construct a deterministic STA S recognizing the trees validated by the DTD. The states of S unify the states of all Glushkov automata and add a unique initial state *I* and a unique final state *F*:

$$\mathit{stat}^S = \uplus_{a\in\Sigma} \mathit{stat}^{G_a} \uplus \{I, F\}$$

The rules of the STA S are obtained systematically from those of the Glushkov automata according to the two following inference schemas:

$$\frac{q_0 \xrightarrow{b} q_1 \in \mathit{rul}^{G_a} \quad q_2 \in \mathit{init}^{G_b} \quad q_3 \in \mathit{fin}^{G_b}}{\begin{array}{c} q_0 \xrightarrow{\mathit{op}\ b:q_0} q_2 \in \mathit{rul}^S \\ q_3 \xrightarrow{\mathit{cl}\ b:q_0} q_1 \in \mathit{rul}^S \end{array}} \qquad \frac{a = \mathit{root} \quad q_0 \in \mathit{init}^{G_a} \quad q_1 \in \mathit{fin}^{G_a}}{\begin{array}{c} I \xrightarrow{\mathit{op}\ a:I} q_0 \in \mathit{rul}^S \\ q_1 \xrightarrow{\mathit{cl}\ a:I} F \in \mathit{rul}^S \end{array}}$$

These schemas can be read as follows. When reading a b-child under an a-node, the STA associates the previous state q_0 of the Glushkov automaton G_a with the b-node, and goes to the initial state q_2 of G_b. Then the children of the b-node are processed in streaming order by the STA. The intended resulting state is the final

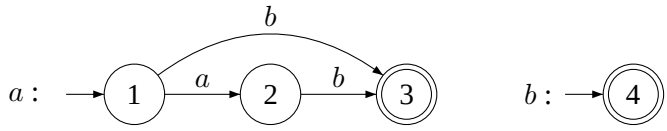


Figure 4.4: Glushkov automata for DTD $a \to ab + b$ and $b \to \epsilon$.

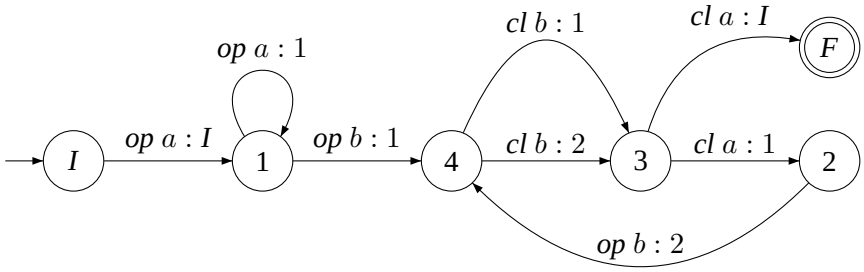


Figure 4.5: The STA for the DTD in Figure 4.4.

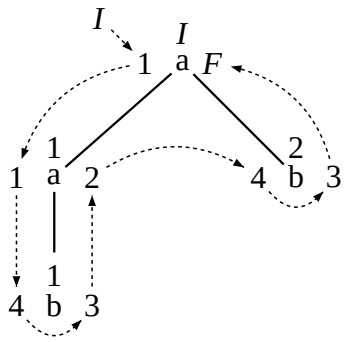


Figure 4.6: Successful run of the STA in Figure 4.5.

state q_3 of G_b. Hence the closing rule for b has q_3 as incoming state, checks that q_0 was associated with the b-node, and goes to the next state q_1 in G_a.

For instance, the STA drawn in Figure 4.5 accepts valid documents for the DTD in Figure 4.4. A successful run on the tree $a(a(b), b)$ is shown in Figure 4.6. This construction preserves determinism, in that DTDs with deterministic Glushkov automata are translated to deterministic STAs. A translation of deterministic restrained competition EDTDs to deterministic ↓TA over the *fcns* encoding is provided by Champavère et al. in [CGLN09] (Lemma 33).

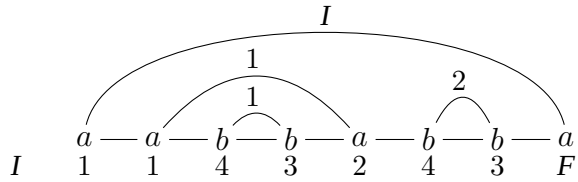


Figure 4.7: Successful run of the NWA in Figure 4.5.

4.4 Nested Word Automata

In this section we present the relation between STAs and nested word automata. This notion of automata is itself very similar to visibly pushdown automata. The difference is in the way the structure is given as input. For visibly pushdown automata, the input word is defined on a visible alphabet, so that each letter is associated with one action (opening or closing, and also the neutral local letters in the general definition). For nested word automata, the input word is given as a flat word plus a binary nesting relation on its positions.

4.4.1 Definition

Nested word automata (NWAs) [Alu07] are equal to STAs syntactically but run on nested words, so they have different semantics. We show that both semantics coincide modulo encoding unranked trees into nested words.

A nested word over Σ is a pair (w, E) where $w \in \Sigma^*$ is a word and $E \subseteq \mathit{dom}(w) \times \mathit{dom}(w)$ a set of forward edges without overlap. We assume that every position in a nested word is adjacent to exactly one edge, and that for every edge, both adjacent positions have the same label.[1]

A run of an NWA A on a nested word (w, E) annotates all positions of $\mathit{dom}(w)$, the start position 0, and all edges in E by states, as illustrated by the example in Figure 4.7. More precisely, a run of A as an NWA consists of two functions $r = (r_\mathrm{e}, r_\mathrm{n})$ with types $r_\mathrm{e} : \mathit{dom}(w) \cup \{0\} \to \mathit{stat}_\mathrm{e}^A$ and $r_\mathrm{n} : E \to \mathit{stat}_\mathrm{n}^A$. It is licensed by A if for all edges $(i, j) \in E$ adjacent to positions labeled by a, the following tuples belong to rul^A:

$$r_\mathrm{e}(i-1) \xrightarrow{\mathit{op}\ a:r_\mathrm{n}(i,j)} r_\mathrm{e}(i)$$
$$r_\mathrm{e}(j-1) \xrightarrow{\mathit{cl}\ a:r_\mathrm{n}(i,j)} r_\mathrm{e}(j)$$

Unranked trees $t \in \mathcal{T}_\Sigma$ can be encoded into nested words $\mathrm{nw}(t) = (w, E)$ over Σ. For instance, the nested word for $a(a(b), b)$ is drawn in Figure 4.7. More formally,

[1] More general definitions of nested words in the literature do permit dangling edges, internal positions, and unmatched labels, that we exclude here.

let $\eta_1 \ \ldots \ \eta_n$ be the sequence of events in t except *start* in their total order. The word:

$$w = a_1 \ldots a_n$$

is the sequence of all $a_i \in \Sigma$ labeling the nodes of event η_i in t where $1 \leq i \leq n$. The edges link opening to closing events of the same node, i.e.:

$$E = \{(i, j) \mid \pi \in \mathit{nod}(t),\ \eta_i = (\mathit{op}, \pi), \eta_j = (\mathit{cl}, \pi)\}$$

4.4.2 Translations into and from STAs

The function $I_{\mathrm{e}} : \mathit{eve}(t) \to \mathit{dom}(\mathrm{nw}(t)) \cup \{0\}$ with $I_{\mathrm{e}}(\mathit{start}) = 0$ and $I_{\mathrm{e}}(\eta_i) = i$ for all $1 \leq i \leq n$ is a bijection, as well as the function $I_{\mathrm{n}} : \mathit{nod}(t) \to E$ with $I_{\mathrm{n}}(\pi) = (I_{\mathrm{e}}((\mathit{op}, \pi)), I_{\mathrm{e}}((\mathit{cl}, \pi)))$. Thus, events of t correspond to positions of $\mathrm{nw}(t)$ or 0 and nodes of t to edges of $\mathrm{nw}(t)$. The edges of t do not have immediate counterparts in $\mathrm{nw}(t)$, but can be inferred from the relations of positions in $\mathrm{nw}(t)$ nevertheless.

Proposition 16. *Let A be an STA over Σ and $t \in \mathcal{T}_\Sigma$ an unranked tree. A run (r_n, r_e) on $\mathrm{nw}(t)$ is licensed by A as an NWA if and only if the run $(r_n \circ I_n, r_e \circ I_e)$ on t is licensed by A as an STA.*

As a consequence, the runs of A on t and $\mathrm{nw}(t)$ correspond bijectively, and t is accepted by A as an STA if and only if $\mathrm{nw}(t)$ is accepted by A as an NWA.

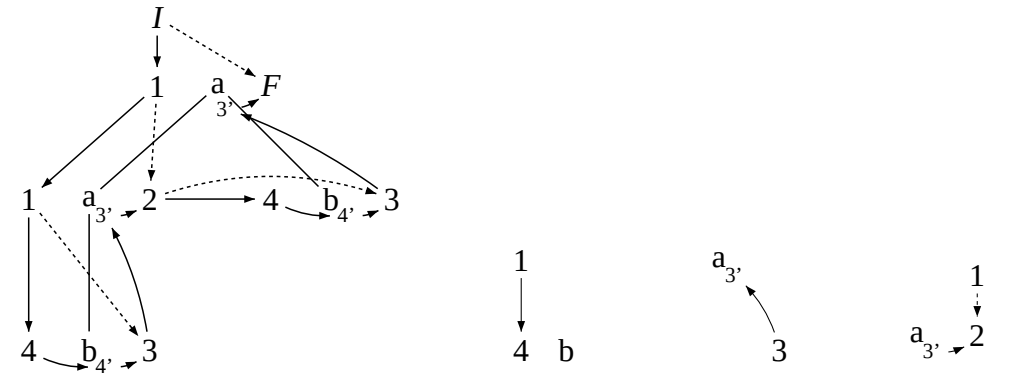
Nested words (w, E) encoding unranked trees satisfy the following restriction:

no hedges: there exists an edge $(1, |w|) \in E$.

Conversely, every nested word satisfying this condition encodes some unranked tree. Every edge (i, j) in E corresponds to one node π of this tree, using the common label of i and j. As no overlap occurs, positions between i and j can be translated into a sequence of trees, defining the children of π. The *no hedges* condition ensures that this sequence of trees has a unique root.

4.5 Pushdown Forest Automata

We recall PFAs from Neumann and Seidl [NS98] which operate on hedges (called forests there), and show how they relate to STAs.



(a) Example of run. (b) *down* $b\ 1 \to 4$ (c) *up* $a\ 3 \to 3'$ (d) *side* $1\ 3' \to 2$

Figure 4.8: Run of a PFA.

4.5.1 Definition

We reformulate the original recursive definition of PFAs evaluators by formalizing a corresponding notion of runs. We restrict ourselves to tree languages, in that we define runs on trees only. This is no serious restriction, since our results extend easily to sequences of trees.

Definition 7. *A* pushdown forest automaton (PFA) *is a tuple* $(\Sigma, \mathit{stat}, \mathit{init}, \mathit{fin}, \mathit{rul})$ *where* Σ *is a finite set,* $\mathit{stat} = \mathit{stat}_e \uplus \mathit{stat}_n$ *is a finite set of states, composed of event states and node states,* $\mathit{init}, \mathit{fin} \subseteq \mathit{stat}_e$ *are finite sets of event states, and rul is a set of rules of the following forms, where* $q_0, q_1 \in \mathit{stat}_e$, $\gamma \in \mathit{stat}_n$ *and* $a \in \Sigma$*:*

$$\mathit{down}\ a\ q_0\ \to\ q_1 \qquad \mathit{side}\ q_0\ \gamma\ \to\ q_1 \qquad \mathit{up}\ a\ q_0\ \to\ \gamma$$

Event states are originally called *forest states* and node states correspond to the original *tree states*. PFAs traverse trees in document order. When leaving a node π, two rules are used. First, an *up*-rule maps the node to some node state. Second, a *side*-rule assigns an event state to the closing event of the node. *up*-rules can be eliminated, but are kept here as in the original definition.

More formally, PFAs P permit runs $r = (r_\mathrm{e}, r_\mathrm{n})$ on trees t, with r_e:$\mathit{eve}(t) \to \mathit{stat}_\mathrm{e}$ and r_n: $\mathit{nod}(t) \to \mathit{stat}_\mathrm{n}$, if P contains the following rules for all nodes $\pi \in \mathit{nod}(t)$ with a label $a \in \Sigma$:

$$\begin{array}{rcl} \mathit{down}\ \ a\ \ r_\mathrm{e}(\mathit{pr}((\mathit{op}, \pi))) & \to & r_\mathrm{e}((\mathit{op}, \pi)) \\ \mathit{side}\ \ r_\mathrm{e}(\mathit{pr}((\mathit{op}, \pi)))\ \ r_\mathrm{n}(\pi) & \to & r_\mathrm{e}((\mathit{cl}, \pi)) \\ \mathit{up}\ \ a\ \ r_\mathrm{e}(\mathit{pr}((\mathit{cl}, \pi))) & \to & r_\mathrm{n}(\pi) \end{array}$$

and $r_{\mathrm{e}}(\mathit{start}) \in \mathit{init}$. The run is successful if $r_{\mathrm{e}}((\mathit{cl}, \epsilon)) \in \mathit{fin}$. Figure 4.8(a) presents a run of a PFA on our example tree. The representation of rules is explained in Figures 4.8(b), 4.8(c) and 4.8(d).

4.5.2 Equivalence to STAs

We present polynomial time translations between weak STAs and PFAs and vice versa, which preserve runs up to simple correspondences and thus languages.

From PFAs to weak STAs

We transform PFAs P into weak STAs $s(P)$ by removing intermediate tree states, identifying rules for *down* and *op*, and combining rules for *up* and *side* into *cl*. Let $\mathit{stat}^{s(P)} = \mathit{stat}^P_{\mathrm{e}}$, $\mathit{init}^{s(P)} = \mathit{init}^P$, and $\mathit{fin}^{s(P)} = \mathit{fin}^P$, and let the following schemas define the rules of $s(P)$:

$$\frac{\mathit{down}\ a\ q_0 \to q_1 \in \mathit{rul}^P}{q_0 \xrightarrow{\mathit{op}\ a:q_0} q_1 \in \mathit{rul}^{s(P)}} \qquad \frac{\begin{array}{c}\mathit{up}\ a\ q_1 \to \gamma_1 \in \mathit{rul}^P \\ \mathit{side}\ q_0\ \gamma_1 \to q_2 \in \mathit{rul}^P\end{array}}{q_1 \xrightarrow{\mathit{cl}\ a:q_0} q_2 \in \mathit{rul}^{s(P)}}$$

From weak STAs to PFAs

Let A be a weak STA. We define a corresponding PFA $p(A)$ such that $s(p(A)) = A$. This shows that $p(A)$ and A recognize the same tree language. Let $\mathit{stat}^{p(A)}_{\mathrm{e}} = \mathit{stat}^A$ and $\mathit{stat}^{p(A)}_{\mathrm{n}} = \Sigma \times \mathit{stat}^A$, initial and final states remaining the same. The following inference schemas detail how the rules of $p(A)$ are inferred from A.

$$\frac{q_0 \xrightarrow{\mathit{op}\ a:q_0} q_1 \in \mathit{rul}^A}{\mathit{down}\ a\ q_0 \to q_1 \in \mathit{rul}^{p(A)}} \qquad \frac{q_0 \xrightarrow{\mathit{cl}\ a:q_1} q_2 \in \mathit{rul}^A}{\begin{array}{c}\mathit{up}\ a\ q_0 \to (a, q_0) \in \mathit{rul}^{p(A)} \\ \mathit{side}\ q_1\ (a, q_0) \to q_2 \in \mathit{rul}^{p(A)}\end{array}}$$

Theorem 2. *Every PFA can be converted into an STA accepting the same language, and vice versa.*

Proof. First, we prove that $L(s(P)) = L(P)$. This translation preserves the first function r_{e} of runs. Since $s(P)$ is weak, this function is sufficient to define a whole run of $s(P)$. Conversely, given a run of $s(P)$ on t, we can easily build the second function r_{n} as every *cl* rule used in r_{e} is generated using an intermediate tree state. These translations preserve acceptance, so $L(P) = L(s(P))$.

Second, we show that for all weak STAs A, $s(p(A)) = A$. Recall that weakness can be assumed w.l.o.g. Translations of *op* and *down* rules are exactly symmetric. The double inclusion of *cl* rules of A and $s(p(A))$ can be easily checked. Initial and final states are also preserved. □

Thus, PFAs can be converted into weak STAs with fewer states so that the tree languages are preserved. Vice versa, there exists a language preserving translation of weak STAs to PFAs which may increase the number of states by a factor of $|\Sigma|$.

The runs of STAs and corresponding PFAs assign the same event states to opening and closing events. This means that they define the same run-based queries, when selecting in event states only. This is illustrated in Figure 4.8(a), by a run of the PFA corresponding to the STA of the previous example Figure 4.6.

As a consequence, we can rely on the query answering algorithm for pushdown forest automata [BS04] for answering run-based weak STA queries. Removing the weakness limitation does not create any problem. This way, we obtain a query answering algorithm for n-ary queries defined by STAs and NWAs.

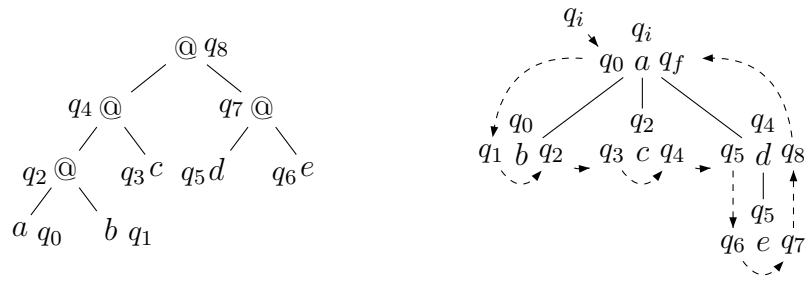
4.6 Standard Tree Automata

In Section 2.1.3, we have seen how standard automata, that were originally defined for ranked trees, can be combined with binary encodings in order to recognize unranked trees. In this section, we consider two of these models. The first one is given by bottom-up tree automata operating on *curry* encodings of trees, also called *Stepwise Tree Automata* [CNT04]. The second one uses top-down tree automata on *fcns* encoding of trees. The reason why we are interested in these models, is that they operate in a way that is compatible with a streaming evaluation. They can be considered as special classes of STAs. We provide back and forth translations between each model and STAs, and show that the translations to STAs preserve determinism. This shows that determinism of STAs is stronger than determinism of these classes.

4.6.1 Stepwise Tree Automata

From Stepwise Tree Automata to STAs

The translation of stepwise tree automata to STAs is quite straightforward, as they can be seen as a weaker form of STAs: a stepwise tree automaton evaluates a hedge (of children of a node) sequentially, from left to right. The difference with STAs is that when evaluating a new tree of the hedge, the state resulting from the evaluation of the beginning of the hedge is unknown. The translation of a stepwise tree automaton A to an STA A' is detailed and proved below, and illustrated in Figure 4.9. The key idea here is to translate an @-rule by a closing rule, that uses the stack to know how the hedge of preceding siblings of the current node was evaluated, and the current state to know what is the state for the subtree rooted at



(a) A run of a stepwise tree automaton A on $t \in \mathcal{T}_{\Sigma_@}$.

(b) A run of the corresponding STA A' on $\mathit{curry}^{-1}(t)$.

Figure 4.9: Example of runs for the translation of stepwise tree automata to STAs.

the current node. Labels are only used at opening.

$$\mathit{stat}^{A'} = \mathit{stat}^{A} \uplus \{q_i, q_f\} \qquad \mathit{init}^{A'} = \{q_i\} \qquad \mathit{fin}^{A'} = \{q_f\}$$

$$\frac{@(q_0, q_1) \to q_2 \in \mathit{rul}^A}{q_1 \xrightarrow{\mathit{cl}\ a:q_0} q_2 \in \mathit{rul}^{A'}} \qquad \frac{a \to q_1 \in \mathit{rul}^A \qquad q_0 \in \mathit{stat}^A}{q_0 \xrightarrow{\mathit{op}\ a:q_0} q_1 \in \mathit{rul}^{A'}} \qquad \frac{q \in \mathit{fin}^A \qquad a \in \Sigma}{q \xrightarrow{\mathit{cl}\ a:q_i} q_f}$$

Correctness relies on the following property, that can be easily proved inductively on the structure of $t \in \mathcal{T}_{\Sigma_@}$:

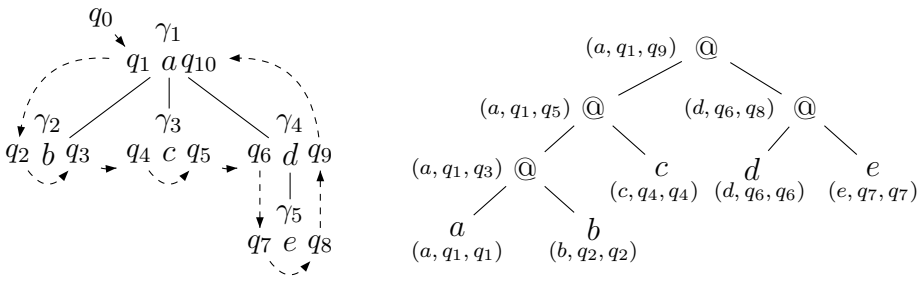
> there is a run r of A on t iff there is a run r' of A' on $\mathit{curry}^{-1}(t)$, and if such runs exist, then $r(\epsilon) = r'((\mathit{cl}, k))$ if the root of $\mathit{curry}^{-1}(t)$ has k children, and $r'((\mathit{op}, \epsilon)) = r(\pi_\epsilon)$ where π_ϵ is the first leaf of t in pre-order.

From STAs to Stepwise Tree Automata

We exhibit a translation from an STA A to a TA recognizing the language of corresponding *curry* encodings of trees, i.e. an equivalent stepwise tree automaton. This time the translation is more intricate, as STAs allow to send the current state from one node to its right sibling, but stepwise tree automata do not. This is why we have to guess this state, and then to check whether this guess corresponds to the state reached when closing the previous sibling. The construction is shown above and illustrated in Figure 4.10.

$$\mathit{stat}^{A'} = \Sigma \times \mathit{stat}^{A} \times \mathit{stat}^{A}$$

$$\frac{q_0 \xrightarrow{\mathit{op}\ a:\gamma} q_1 \in \mathit{rul}^A \qquad q_2 \xrightarrow{\mathit{cl}\ a:\gamma} q_3 \in \mathit{rul}^A \qquad q_0 \in \mathit{init}^A \qquad q_3 \in \mathit{fin}^A}{(a, q_1, q_2) \in \mathit{fin}^{A'}}$$



(a) A run of the STA A on $t \in \mathcal{T}_\Sigma$. (b) A run of the corresponding stepwise tree automaton A' on $\mathit{curry}(t)$.

Figure 4.10: Example of runs for the translation of STAs to stepwise tree automata.

$$\frac{q_0 \xrightarrow{\mathit{op}\ a:\gamma} q_1 \in \mathit{rul}^A}{a \to (a, q_1, q_1) \in \mathit{rul}^{A'}}$$

$$\frac{q_0 \xrightarrow{\mathit{op}\ b:\gamma} q_1 \in \mathit{rul}^A \qquad q_2 \xrightarrow{\mathit{cl}\ b:\gamma} q_3 \in \mathit{rul}^A \qquad q_4 \in \mathit{stat}^A \qquad a \in \Sigma}{@((a, q_4, q_0), (b, q_1, q_2)) \to (a, q_4, q_3) \in \mathit{rul}^{A'}}$$

The following invariant can be proved inductively on the structure of $t \in \mathcal{T}_{\Sigma_@}$:

> there is a run r' of A' on t such that $r'(\epsilon) = (a, q_0, q_1)$ iff the root of $\mathit{curry}^{-1}(t)$ is labeled by a, there is a run r of A on $\mathit{curry}^{-1}(t)$ such that $r((\mathit{op}, \epsilon)) = q_0$ and $r((\mathit{cl}, k)) = q_1$ where k is the last child of the root.

4.6.2 Top-Down Tree Automata w.r.t. *fcns* Encoding

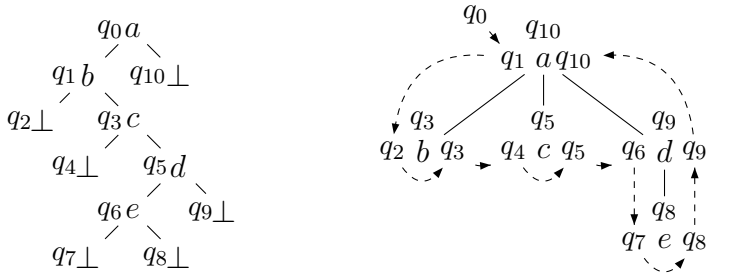
As already mentioned in Section 2.2, DTDs can easily be translated into TAs over *fcns* encodings of trees. We now relate these automata to STAs.

From Top-Down Tree Automata to STAs

Let A be a $\downarrow$TA recognizing binary trees in $\mathcal{T}_{\Sigma_\perp}$, that are *fcns*-encodings of unranked trees. We define an STA A' over Σ such that $L(A) = L(A')$. This is illustrated by Figure 4.11, with runs of A on $\mathit{fcns}(t)$ and A' on t.

$$\begin{aligned} \mathit{stat}^{A'} &= \mathit{stat}^A \\ \mathit{init}^{A'} &= \mathit{init}^A \\ \mathit{fin}^{A'} &= \mathit{stat}^A \end{aligned} \qquad \frac{q, a \to (q_1, q_2) \in \mathit{rul}^A}{q \xrightarrow{\mathit{op}\ a:q_2} q_1 \in \mathit{rul}^{A'}} \qquad \frac{\perp \to q_1 \in \mathit{rul}^A \quad a \in \Sigma \quad q_2 \in \mathit{stat}^A}{q_1 \xrightarrow{\mathit{cl}\ a:q_2} q_2 \in \mathit{rul}^{A'}}$$

This preserves determinism, and the correctness is easily proved using the following invariant:



(a) A run of a ↓TA A on $t \in \mathcal{T}_{\Sigma_\perp}$.

(b) A run of the corresponding STA A' on $\mathit{fcns}^{-1}(t)$.

Figure 4.11: Example of runs for the translation of ↓TAs over *fcns* encoding to STAs.

> if $h = (t_1, \ldots, t_k)$ is an hedge over Σ, then there is a run r of A on $\mathit{fcns}_{\mathcal{H}}(h)$ iff there is a run r' of A' on h, and if such runs exist, then, if π' is the root of t_1 and π the corresponding node in $\mathit{fcns}_{\mathcal{H}}(h)$ we have: $r'_{\mathrm{e}}((\mathit{op}, \pi')) = r(\pi{\cdot}1)$ and $r'_{\mathrm{n}}(\pi') = r(\pi{\cdot}2)$.

From STAs to Top-Down Tree Automata

Let A be an STA over the alphabet Σ. We define the ↓TA A' over $\Sigma_\perp$ such that $L(A') = L(A)$:

$$\mathit{stat}^{A'} = \mathit{stat}^{A} \times \mathit{stat}^{A} \qquad \mathit{fin}^{A'} = \mathit{init}^{A} \times \mathit{fin}^{A}$$

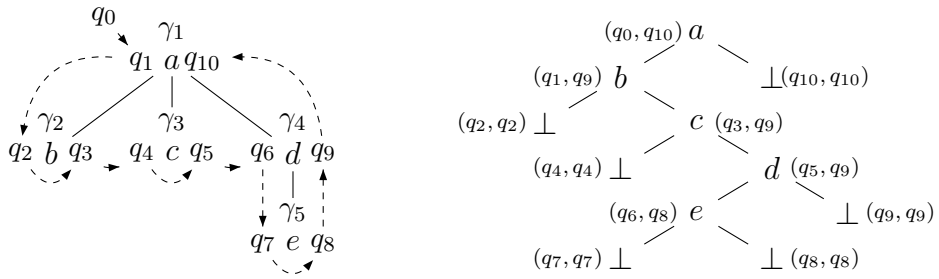
$$\frac{q_0 \xrightarrow{\mathit{op}\ a:\gamma} q_1 \in \mathit{rul}^A \qquad q_2 \xrightarrow{\mathit{cl}\ a:\gamma} q_3 \in \mathit{rul}^A \qquad q_4 \in \mathit{stat}^A}{(q_0, q_4), a \to ((q_1, q_2), (q_3, q_4)) \in \mathit{rul}^{A'}} \qquad \frac{q \in \mathit{stat}^A}{\perp \to (q, q) \in \mathit{rul}^{A'}}$$

Figure 4.12 illustrates this translation. The following property is easy to prove by induction on the structure of t, and gives the main idea of the construction:

> there is a run r' of A' on t iff there is a run r of A on the hedge $\mathit{fcns}^{-1}(t)$, and if such runs exist then $r'(\epsilon) = (q_0, q_1)$ iff there is a run of A on $\mathit{fcns}^{-1}(t)$ starting in q_0 and ending in q_1.

4.7 Conclusion

These translations between automata models allow to reuse algorithms designed for specific models. In our framework, automata can be used for schema definition or query definition. While STAs, NWAs and PFAs are quite similar models,



(a) A run of the STA A on $t \in \mathcal{T}_\Sigma$. (b) A run of the corresponding ↓TA A on *fcns*(t).

Figure 4.12: Example of runs for the translation of STAs to TAs over *fcns* encoding.

operating in pre-order traversals of trees, the use of binary encodings on top of ranked tree automata define models with weaker notions of determinism.

In the remainder of the manuscript, we use dSTAs for defining queries and schemas. STAs benefit from a simple definition, which implementation (using SAX, for instance) is easy to explain. Moreover, STAs are closely related to our computational model. An STA can be implemented by an SRAM where the working tape stores the current configuration, i.e. the current node, and the stack of node states for its ancestors. The next chapter provides an example of how an algorithm can be defined on top of STAs.

Chapter 5

Earliest Query Answering for Streaming Tree Automata

Contents

5.1 Introduction

Streamability of queries defined by deterministic automata is investigated in this chapter. We prove that queries defined by dSTAs, when restricted to shallow trees, are m-streamable for all $m \in \mathbb{N}_0$. They are however not ∞-streamable, as queries with high concurrency can be defined with small dSTAs. In order to obtain these

results, we propose an earliest query answering algorithm, for queries defined by dSTAs.

Earliest Query Answering (EQA) has been introduced by Bar-Yossef et al. in [BYFJ05] and Berlea in [Ber06]. An EQA algorithm outputs selected (tuples of) nodes at the earliest time point when they can be output. Symmetrically, it rejects failed candidates at the earliest time point, once no valid continuation of the stream will select them. Violating one of these constraints means that some candidate is unnecessarily buffered. Indeed, EQA algorithms only memorize alive candidates. This corresponds to a lower memory bound for our computational model, as already proved in Proposition 7.

In this chapter, we present an EQA algorithm for dSTAs queries. As previously mentioned, EQA ensures good properties in terms of space complexity. Thanks to determinism, our algorithm is also efficient in terms of time cost. When the depth of valid trees is bounded, this algorithm achieves a PTIME preprocessing, and then a PTIME cost per event and per candidate, in the size of dSTAs defining the query and schema. The main idea of the algorithm is the dynamic computation of safe states, that ensure selection (resp. rejection) of candidates.

The complexity of EQA is also investigated, for arbitrary query languages. Deciding for selection and rejection in an earliest manner is often computationally hard, and can be reduced to inclusion of Boolean queries. As a consequence, for non-deterministic STAs, earliest selection and rejection is EXPTIME-complete. Thus, there is no PTIME EQA algorithm for queries by STAs. For XPath, we exhibit a fragment with only downward axes, for which EQA is not feasible in PTIME, unless PTIME =NP.

Related work The idea of earliest query answering originates from two papers. In [BYFJ05], Bar-Yossef et al. define the concurrency of a query w.r.t. a tree, and prove that it constitutes a lower memory bound for a fragment of XPath. They also provide an algorithm with space complexity close to the concurrency for shallow trees. In [Ber06], Berlea proposes an EQA algorithm for queries defined by grammars, and then translated into pushdown forest automata. This algorithm is however different from ours, as it assumes an infinite alphabet and does not take schemas into account. This is a major difference, as explained in Section 1.4.

Earliest detection of rejected candidates is also studied by Benedikt et al. in [BJLW08] for filtering XML streams, through the *fast-fail* property. The authors prove that this problem is not tractable unless PTIME = PSPACE. The solution adopted by the authors is to approximate the detection of rejected candidates.

In the streaming literature, it is often claimed that answers are output as soon as possible. From the hardness results previously mentioned, this is often false. For instance Gou and Chirkova [GC07a] claim that their algorithm *achieves op-*

timal buffering-space performance on a fragment of XPath that contains tree patterns. Their algorithm runs in PTIME, which is impossible for EQA algorithms, unless PTIME = NP. Usually, a query answering algorithm for XPath outputs an answer when all positive filters have found a match, and the current event is outside the scopes of all negative filters. This is the case for instance for SPEX, proposed by Olteanu in [Olt07b] and for the logics considered by Benedikt and Jeffrey in [BJ07]. These algorithms are not earliest, because it could be decided before the end of the scopes of negative filters whether they can still be satisfied in any continuation of the stream. Consider for instance the XPath expression $//a[b \text{ or not}(b)]$ that selects all a-nodes, if they have a b-child or not. Here, all a-nodes can be selected when they are read, as the filter is always true. However these algorithms will output a-nodes when closing them.

Madhusudan and Viswanathan [MV08] propose an EQA algorithm for n-ary queries defined by non-deterministic nested word automata recognizing canonical languages, without schema considerations. However, the authors assume that the input automaton does not accept the full linearization of a tree, but the smallest prefix of a tree linearization such that all well-nested suffixes are in the canonical language of the query. Transforming an automaton recognizing a canonical language, to an equivalent one accepting these prefixes is a complex task. Our algorithm avoids its entire construction by computing its rules on demand. Another difference is that we require deterministic automata. In the non-deterministic case, the complexity of this transformation is not studied by Madhusudan and Viswanathan.

Earliest Query Answering algorithms decide at every event the safety of outputting (resp. rejecting) every candidate. This safety property seems related to safety properties studied in formal verification, where the system has to verify such a property in every possible future. For instance in [KV01], Kupferman and Vardi propose to build an automaton recognizing all bad prefixes, such that all suffixes will lead the system into a bad configuration. The links between such formal verification methods and earliest query answering are still to be investigated.

5.2 Earliest Query Answering

We recall the foundations of earliest query answering (EQA). In Section 3.4, we introduced the notions of safe selection and rejection: A tuple τ is safely selected (resp. rejected) by a query at event η if τ is selected (resp. rejected) in all valid continuations of the stream beyond η. We formalize these notions through sufficient events for selection and rejection, and derive some decision problems of EQA algorithms for n-ary node selection queries. We establish lower complexity bounds for such algorithms.

5.2.1 Earliest Event for Selection

Before defining earliest events for selection, we introduce *sufficient events for selection*. Let $t \in \mathcal{T}_\Sigma$ be an unranked tree and Q a query of arity n. An event $\eta \in \mathit{eve}(t)$ is said sufficient for the selection of tuple $\tau \in \mathit{nod}(t)^n$ by Q if for every continuation of the stream beyond η, τ is selected by Q. This quantification over all continuations is expressed through all trees sharing the same prefix until η, in the following definition. Note that this formalizes the notion of safety for selection (resp. rejection) briefly introduced in Section 3.4.

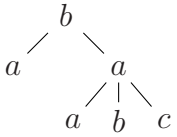
Definition 8 (Sufficient events for selection)**.** *Let Q be an n-ary query over Σ and $t \in \mathit{dom}(Q)$ a tree. We relate tuples $\tau \in \mathit{nod}(t)^n$ to events $\eta \in \mathit{eve}(t)$ that are sufficient for their selection:*

$$(\tau, \eta) \in \mathit{sel}_Q(t) \;\Leftrightarrow\; \begin{cases} \tau \in \mathit{dom}_\eta(t)^n \wedge \\ \forall t' \in \mathit{dom}(Q).\ \mathit{equal}_\eta(t, t') \Rightarrow \tau \in Q(t') \end{cases}$$

The first condition, $\tau \in \mathit{dom}_\eta(t)^n$, restricts the considered tuples to those containing nodes that were read before η, as streaming algorithms cannot output nodes that have not be seen yet. Note that $(\tau, \eta) \in \mathit{sel}_Q(t)$ implies $\tau \in Q(t)$. Furthermore, successors of sufficient events are sufficient.

The *earliest* event η for selecting τ is the first sufficient event for selecting τ:

$$(\tau, \eta) \in \mathit{earliest\_sel}_Q(t) \Leftrightarrow \eta = \min_{\preceq}\{\eta' \mid (\tau, \eta') \in \mathit{sel}_Q(t)\}$$



Consider for instance the monadic query Q_1 with schema $\mathcal{T}_{\{a,b,c\}}$ defined by the XPath expression $/\mathit{ch}^*::a[\mathit{ch}::c]/\mathit{ch}::b$, or equivalently by the first-order formula $\mathit{lab}_b(x) \wedge \exists y.\ (\mathit{lab}_a(y) \wedge \mathit{ch}(y, x) \wedge \exists z.\ (\mathit{ch}(y, z) \wedge \mathit{lab}_c(z)))$ with one free variable x. On the tree $t = b(a, a(a, b, c))$, the earliest time point to select node $2{\cdot}2$ is event $(\mathit{op}, 2{\cdot}3)$ when the c-child is opened, i.e., $((2{\cdot}2), (\mathit{op}, 2{\cdot}3)) \in \mathit{earliest\_sel}_{Q_1}(t)$. Events following $(\mathit{op}, 2{\cdot}3)$ are sufficient for selecting $2{\cdot}2$, but not earliest. For instance: $((2{\cdot}2), (\mathit{cl}, 2{\cdot}3)) \in \mathit{sel}_{Q_1}(t) - \mathit{earliest\_sel}_{Q_1}(t)$.

For query Q_2 defined by the same XPath expression, but with the more restrictive schema, requiring that all inner a-nodes have at least one c-child, we can select node $2{\cdot}2$ at opening time, i.e., $((2{\cdot}2), (\mathit{op}, 2{\cdot}2)) \in \mathit{earliest\_sel}_{Q_2}(t)$.

5.2.2 Earliest Event for Rejection

For optimal memory management, it is equally important to discard *rejected* answer candidates in an earliest manner, i.e., candidates that will never be selected

in any possible future. Going one step further, one might also want to remove rejected partial candidates, for which no completion will ever be selected in any future.

Definition 9 (Sufficient events for rejection)**.** *We call a candidate τ* rejected *at event η, or equivalently η sufficient for rejecting τ, if no completion of τ can be selected in the future:*

$$(\tau,\eta) \in \mathit{rej}_Q(t) \;\Leftrightarrow\; \begin{cases} \tau \in \mathit{dom}^{\bullet}_{\eta}(t)^n \wedge \\ \forall t' \in \mathit{dom}(Q).\ \mathit{equal}_{\eta}(t,t') \Rightarrow \\ \qquad \forall \tau' \in \mathit{compl}(\tau,t',\eta).\ \tau' \notin Q(t') \end{cases}$$

The *earliest* event η for rejecting τ is the first sufficient event for rejecting τ:

$$(\tau,\eta) \in \mathit{earliest\_rej}_Q(t) \Leftrightarrow \eta = \min_{\preceq}\{\eta' \mid (\tau,\eta') \in \mathit{rej}_Q(t)\}$$

We illustrate these definitions at the query Q_1 defined by the XPath expression $/\mathit{ch}^*::a[\mathit{ch}::c]/\mathit{ch}::b$, on the tree $t = b(a, a(a, b))$. All nodes π that are not labeled by b (and the root ϵ) can be immediately rejected, i.e. $((\pi),(\mathit{op},\pi)) \in \mathit{earliest\_rej}_{Q_1}(t)$. For the b-node $2\cdot 2$, the earliest event for rejection is $(\mathit{cl}, 2)$, as all siblings of $2\cdot 2$ must have been inspected.

Link to Concurrency Earliest events for selection and rejection are closely related to the concurrency of the query, introduced in Section 3.2.3. A tuple τ is alive at event η iff η is not sufficient for selecting τ, nor for rejecting it:

$$(\tau,\eta) \in \mathit{alive}_Q(t) \quad \Leftrightarrow \quad (\tau,\eta) \notin \mathit{sel}_Q(t) \cup \mathit{rej}_Q(t)$$

5.3 Complexity of Selection Sufficiency

5.3.1 Sufficiency Problem

The definition of sufficient events for selection leads to the problem of deciding whether an event η is sufficient for selecting a tuple τ. This problem has to be solved by all EQA algorithms at every processed event, and hence will give us lower bounds for the per-event time of EQA algorithms. For simplicity, we only address the sufficiency for selection here, not for rejection.

Definition 10 (Sufficiency problem)**.** *The* SUFFICIENCY *problem is defined by the following parameters, input and outputs:*

PARAMETERS: a signature Σ, a class $\mathcal{Q}$ of queries of arity n,

INPUTS: an expression $e \in \mathcal{Q}$, a tree $t \in \mathcal{T}_\Sigma$, an n-tuple $\tau \in nod(t)^n$, and an event $\eta \in eve(t) - \{start\}$.

OUTPUT: the truth value of $(\tau, \eta) \in sel_{Q_e}(t)$.

We provide hardness results for SUFFICIENCY. To establish these results, we reduce language inclusion to SUFFICIENCY.

5.3.2 Reduction from Language Inclusion

Let $c_{Q,\tau,\eta,t}$ be the set of trees on which τ is selected or that have a prefix different from $t^{\preceq\eta}$:

$$c_{Q,\tau,\eta,t} = \{t' \in \mathcal{T}_\Sigma \mid equal_\eta(t, t') \Rightarrow \tau \in Q(t')\}$$

Then we can rephrase sufficiency for selection in the following way.

Lemma 3. $(\tau, \eta) \in sel_Q(t) \quad \Leftrightarrow \quad \tau \in dom_\eta(t)^n \wedge dom(Q) \subseteq c_{Q,\tau,\eta,t}$

This reformulation relates SUFFICIENCY to language inclusion for classes of Boolean queries. The INCLUSION problem for a class $\mathcal{Q}$ of Boolean queries inputs an expression $e \in \mathcal{Q}$ and outputs the truth value of $dom(Q_e) \subseteq L_{Q_e}$. UNIVERSALITY returns the truth value of $\mathcal{T}_\Sigma \subseteq L_{Q_e}$ instead.

Lemma 4 (Hardness)**.** *For all classes $\mathcal{Q}$ of Boolean queries there is a linear time reduction of* INCLUSION *to* SUFFICIENCY*, and of* UNIVERSALITY *to* SUFFICIENCY *for queries with schema $\mathcal{T}_\Sigma$.*

Proof. Let $e \in \mathcal{Q}$ and $t \in dom(Q)$ a tree. Since Q_e is Boolean, the definition yields $c_{Q_e,(),start,t} = L_{Q_e}$. Thus, Lemma 3 proves that $((), start) \in sel_{Q_e}(t)$ if and only if $dom(Q_e) \subseteq L_{Q_e}$. □

5.3.3 Hardness of EQA for XPath and STAs

We consider Boolean filters in the following fragment of Forward XPath, where $\ell \in \Sigma \cup \{*\}$:

$$F ::= [ch::\ell\, F] \mid [ch^*::\ell\, F] \mid [F_1\ and\ F_2] \mid [not(F)] \mid [true]$$

Proposition 17. SUFFICIENCY *for Boolean queries defined in the above fragment of Forward XPath is coNP-hard, even without schema assumptions.*

Proof. According to Lemma 4, SUFFICIENCY without schemas is harder than UNIVERSALITY of Boolean queries. The latter problem was proven coNP-hard for the above fragment of Forward XPath in Proposition 4. □

Adding schemas does not reduce the complexity of the problem. As a consequence, every EQA algorithm for a larger fragment of XPath cannot be in polynomial time, except if PTIME $=$ NP.

For queries defined by non-deterministic automata, SUFFICIENCY remains hard, even with Boolean queries.

Proposition 18. SUFFICIENCY *for Boolean queries defined by STAs is* EXPTIME*-hard.*

Proof. By Lemma 4, SUFFICIENCY without schemas is harder than UNIVERSALITY for STAs, and thus EXPTIME-hard by Proposition 15. □

However, when restricted to deterministic STAs, the problem becomes tractable. The crucial point here is that dSTAs can check equality of prefixes of two trees until event η deterministically.

As previously introduced, we write Q_A for the query defined by the STA A recognizing a canonical language, i.e., $L_{Q_A} = L(A)$ and $\mathit{dom}(Q_A) = \mathcal{T}_\Sigma$. When a schema is provided by an STA B, $Q_{A,B}$ denotes the query such that $L_{Q_{A,B}} = L(A)$ and $\mathit{dom}(Q_{A,B}) = L(B)$.

Lemma 5. *If a dSTA A recognizes a canonical language, then for all $t \in \mathcal{T}_\Sigma$, $\tau \in \mathit{nod}(t)^n$ and $\eta \in \mathit{eve}(t)$, we can compute a dSTA recognizing the language $c_{Q_A,\tau,\eta,t}$ in* PTIME *in $|A|$, $|t|$, $|\tau|$ and $|\eta|$.*

Proof. We prove that we can build a dSTA recognizing $c_{A,\tau,\eta,t}$ in polynomial time from A, t, $\pi \in \mathit{nod}(t)$, $\alpha \in \{\mathit{op}, \mathit{cl}\}$, and $\tau \in \mathit{nod}(t)^n$. We define two tree languages:

$$\mathit{Eq}_{t,\eta} = \{t' \mid \mathit{equal}_\eta(t, t')\} \qquad Q_\tau = \{t' \mid \tau \in Q_A(t')\}$$

With these definitions, we get $c_{Q_A,\tau,\eta,t} = \mathit{Eq}^{\text{compl}}_{t,\eta} \cup Q_\tau$ where $L^{\text{compl}} = \{t \in \mathcal{T}_\Sigma \mid t \notin L\}$ for $L \subseteq \mathcal{T}_\Sigma$. Hence it suffices to build dSTAs recognizing $\mathit{Eq}_{t,\eta}$ and Q_τ in PTIME.

First of all, we define a weak dSTA recognizing $\mathit{Eq}_{t,\eta} = \{t' \mid \mathit{equal}_\eta(t, t')\}$. We set $\mathit{stat}_{\text{e}} = \mathit{eve}(t^{\preceq\eta})$, $\mathit{stat}_{\text{n}} = \{\gamma\}$ (arbitrary), $\mathit{init} = \{\mathit{start}\}$, $\mathit{fin} = \{\eta\}$, and the following rules where $\preceq$ and pr are interpreted on $\mathit{eve}(t)$:

$$\frac{(\alpha,\pi) \preceq \eta \qquad a = \mathit{lab}^t(\pi)}{\mathit{pr}((\alpha,\pi)) \xrightarrow{\alpha\ a:\gamma} (\alpha,\pi)} \qquad \frac{a \in \Sigma}{\eta \xrightarrow{\mathit{op}\ a:\gamma} \eta \qquad \eta \xrightarrow{\mathit{cl}\ a:\gamma} \eta}$$

Second, we define a dSTA recognizing the set $Q_\tau = \{t' \mid \tau \in Q_A(t')\}$. Such a dSTA can be built in several steps. We first build a dSTA A' recognizing all trees annotated with the tuple τ, i.e.:

$$L(A') = \{t * \tau \mid t \in \mathcal{T}_\Sigma\}$$

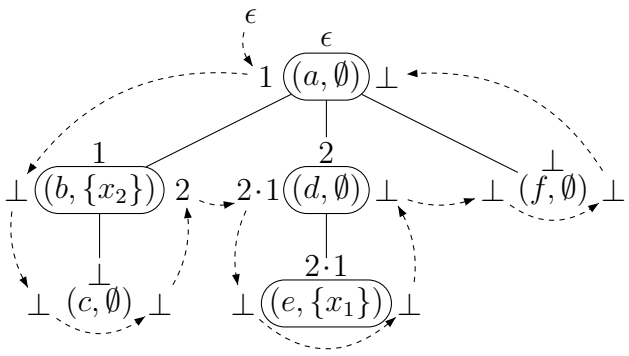


Figure 5.1: A run of the dSTA A', when $\tau = (2{\cdot}1, 1)$. The domain for this τ is *domain* $= \{\epsilon, 1, 2, 2{\cdot}1\}$, as indicated by framed nodes.

Then we can intersect A' with A, in order to distinguish all annotated trees on which τ is selected by Q_A. Finally, we can project on the Σ-component in order to obtain the desired trees:

$$Q_\tau = \Pi_\Sigma(Q_A \wedge Q_{A'})$$

The corresponding automata operations preserve determinism, in this particular case: for each tree $t \in \mathcal{T}_\Sigma$, there is at most one run of $A \cap A'$ on $t * \tau$, as both automata are deterministic. Hence, after projection, there is also at most one run on t, and thus the determinism is preserved by the projection, in this case.

It remains to detail the construction of A'. If the arity of Q_A is $n = 0$ then $\tau = ()$ and we can take a universal automaton, as $L(A') = \mathcal{T}_\Sigma$. Otherwise, in order to define this automaton in polynomial size in $|\tau|$, some preprocessing on τ is required, which factorizes common prefixes of node addresses. Roughly speaking, we call *domain* the domain of the smallest tree containing τ, and build a dSTA that computes in its states the next element of *domain* to be checked, as illustrated in Figure 5.1. Formally, let *domain* be the set of positions π smaller or equal to some position of τ for the order defined by $\pi.i < \pi.j$ if $i < j$ and $\pi < \pi.i$. We write $\mathit{domain}_\perp = \mathit{domain} \cup \{\perp\}$. We introduce the function *next*: $\{\mathit{op}, \mathit{cl}\} \times (\mathbb{N}^* \cup \{\perp\}) \to \mathit{domain}_\perp$ that indicates whether the domain still continues above (resp. at the right of) the current node π, when called with (op, π) (resp. (cl, π)):

$$\begin{cases} \mathit{next}(\mathit{op}, \pi) = \pi{\cdot}1 & \text{if } \pi{\cdot}1 \in \mathit{domain}, \quad \perp \text{ otherwise} \\ \mathit{next}(\mathit{cl}, \pi{\cdot}i) = \pi{\cdot}(i+1) & \text{if } \pi{\cdot}(i+1) \in \mathit{domain}, \perp \text{ otherwise} \\ \mathit{next}(\alpha, \perp) = \perp & \text{for } \alpha \in \{\mathit{op}, \mathit{cl}\} \end{cases}$$

We also introduce the function vars_τ: $\mathit{domain}_\perp \to 2^{\mathcal{V}_n}$ that associates with each

node the variables corresponding to the annotation by τ:

$$\text{if } \tau = (\pi_1, \ldots, \pi_n) \text{ then } \begin{cases} \mathit{vars}_\tau(\pi) = \{x_i \mid \pi_i = \pi\} \\ \mathit{vars}_\tau(\bot) = \emptyset \end{cases}$$

We can now define the dSTA A'. A run of A' is shown in Figure 5.1.

$$\mathit{stat}_{\mathrm{e}}^{A'} = \mathit{stat}_{\mathrm{n}}^{A'} = \mathit{domain}_\bot \qquad \mathit{init}^{A'} = \epsilon \qquad \mathit{fin}^{A'} = \{\bot\}$$

$$\frac{a \in \Sigma \qquad \pi, \pi' \in \mathit{domain}_\bot \qquad l = \mathit{vars}_\tau(\pi)}{\begin{array}{c} \pi \xrightarrow{\mathit{op}\ (a,l):\pi} \mathit{next}(\mathit{op}, \pi) \in \mathit{rul}^{A'} \\ \pi' \xrightarrow{\mathit{cl}\ (a,l):\pi} \mathit{next}(\mathit{cl}, \pi) \in \mathit{rul}^{A'} \end{array}}$$

□

Theorem 3. SUFFICIENCY *for n-ary dSTA queries is in polynomial time.*

Proof. We can test $L(B) \subseteq c_{A,\tau,\eta,t}$ in polynomial time, if B is given an dSTA, since we can compute a dSTA for $c_{A,\tau,\eta,t}$ in linear time by Lemma 5, and since INCLUSION for dSTAs is in polynomial time (Proposition 15). □

As a corollary SUFFICIENCY for STAs is EXPTIME-complete. A EXPTIME algorithm follows from STA determinization and Theorem 3. By Proposition 18, the lower bound holds already for STAs defining Boolean queries.

5.4 EQA Algorithm for dSTAs

From the previous results, we know that SUFFICIENCY can be decided in PTIME for queries defined by dSTAs. In this section we propose an earliest query answering algorithm for such queries, using polynomial per-event time and space for each candidate. We start with a static transformation of the dSTA A defining the query Q_A into another dSTA $E(A)$, in Section 5.4.1. $E(A)$ and A recognize the same language, but the states of $E(A)$ contain enough information for deciding sufficiency for selection and rejection. This is not the case for A, as in general the sufficiency depends on the configuration, and hence from the states of the ancestor nodes (as their states will be later used at closing). However, this translation of A into $E(A)$ implies an exponential blow-up. In Section 5.4.2, we propose a PTIME algorithm that avoids this blow-up by constructing the needed parts of $E(A)$ on the fly. In Section 5.4.3, we show how schemas can be taken into account, and illustrate it at an example in Section 5.4.4. Finally, we show how the algorithm can be efficiently implemented in Section 5.4.5.

5.4.1 Safe States Computation for dSTAs

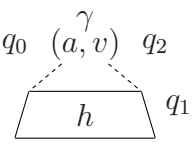
We define a partial run *r* of an STA A on a tree t like a run, except that it operates only on a prefix $t^{\preceq\eta}$ for some event $\eta \in \mathit{eve}(t)$. We write $\mathit{p\_runs}^A(t)$ for the set of all partial runs of A on t.

Safe States for Selection

Let A be a dSTA over $\Sigma \times 2^{\mathcal{V}_n}$ defining a query Q_A, $t \in \mathcal{T}_\Sigma$, $\eta \in \mathit{eve}(t)$, and $\tau \in \mathit{nod}(t)^n$. We consider for the moment queries with universal schemas.

Definition 11 (safe states for selection)**.** *We call a state $q \in \mathit{stat}_e^A$ safe for selection of τ at event η if the existence of a partial run r of A on t that maps η to q implies $(\tau, \eta) \in \mathit{sel}_{Q_A}(t)$. In other terms, these are the states that ensure sufficiency for selection when they are reached:*

$$\mathit{safe\_sel}^A_{(\tau,\eta)}(t) = \{q \mid (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_e(\eta) = q) \Rightarrow (\tau, \eta) \in \mathit{sel}_{Q_A}(t)\}$$



In general, A does not have safe states, or more precisely, a sufficient event can be reached by a run of A, but the corresponding run does not go into a safe state for selection. We now describe how these states can be computed by a new dSTA $E(A)$, which permits to decide sufficiency. Here we need some auxiliary definitions. Let $\mathit{runs}^A_{q_0 \to q_1}(h)$ be the set of runs of an STA A on a hedge h that start in state q_0 and end in state q_1. The operator $\mathit{ev\_cl}^A(h, q_0, (a, v), \gamma)$ evaluates hedge h from state q_0 and subsequently applies a closing rule with label $(a, v) \in \Sigma \times 2^{\mathcal{V}_n}$ and state γ:

$$\mathit{ev\_cl}^A(h, q_0, (a, v), \gamma) = \{q_2 \mid \exists r \in \mathit{runs}^A_{q_0 \to q_1}(h).\ q_1 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^A\}$$

We consider continuations through hedges in $\mathcal{H}_{\mathit{sel}} = \mathcal{H}_{\Sigma \times \{\emptyset\}}$, as safe states for selection are defined for complete tuples, and thus valid continuations cannot use variables anymore. The operator $\mathit{univ\_sel}^A((a, v), \gamma, P)$ computes all states, from where all hedges in $\mathcal{H}_{\mathit{sel}}$ can be evaluated and closed w.r.t. (a, v) and γ into a state of $P \subseteq \mathit{stat}^A_{\mathrm{e}}$:

$$\mathit{univ\_sel}^A((a, v), \gamma, P) = \{q_0 \mid \forall h \in \mathcal{H}_{\mathit{sel}}.\ \mathit{ev\_cl}^A(h, q_0, (a, v), \gamma) \cap P \neq \emptyset\}$$

Given A, t, and τ, we can compute inductively the safe states $S_{\mathit{sel}}(\eta) = \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$ for all events $\eta \in \mathit{eve}(t)$, using three propagation rules, as illustrated in Figure 5.2 and proved by Lemma 6.

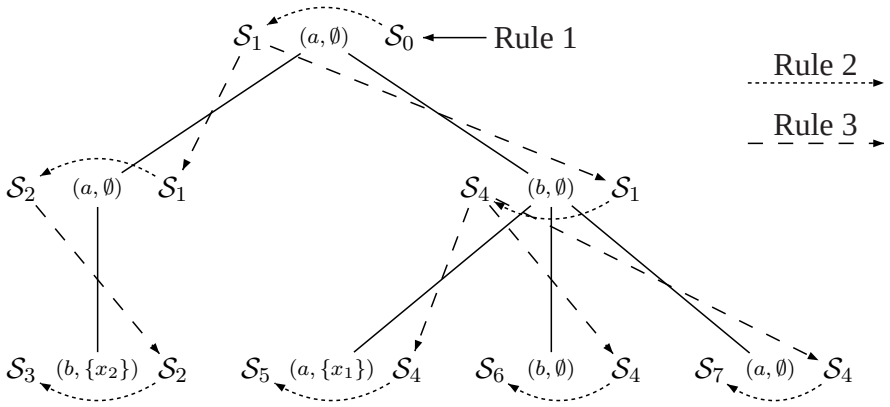


Figure 5.2: Propagation rules for safe states.

Rule 1 For the closing event of the root, the event (cl, ϵ) is sufficient for selection of the given τ on t iff all continuations after (cl, ϵ) succeed. The only existing continuation is the empty one, so the sufficiency only depends on the success of the run. Thus when closing the root, the set of safe states for selection are the final states:

$$S_{\mathit{sel}}((\mathit{cl}, \epsilon)) = \mathit{fin}^A$$

Rule 2 At each node π, the safe states for the opening event can be computed from those of the corresponding closing event. These are the states for which the traversal of any hedge h (of children), followed by the closure of the node, leads to a safe state at closing.

$$S_{\mathit{sel}}((\mathit{op}, \pi)) = \mathit{univ\_sel}^A((a, v), \gamma, S_{\mathit{sel}}((\mathit{cl}, \pi)))$$

where $(a, v) = \mathit{lab}^t(\pi)$ and $\gamma = r^A_{\mathrm{n}}(\pi)$.

Rule 3 Third, the safe states for the opening event of π are equal to those for the closing events of children of π:

$$S_{\mathit{sel}}((\mathit{cl}, \pi \cdot i)) = S_{\mathit{sel}}((\mathit{op}, \pi))$$

This might seem surprising at first sight. However, the condition for rule 2 can be rephrased in the following way for rule 3: the traversal of any hedge (here, of right siblings and their descendants) followed by the closure of the parent node must lead to a safe state for closing the parent node.

Safe States for Rejection

The treatment of safe states for rejection is more delicate. Here we have to assume determinism and completeness for a proper treatment of partial candidates. The definitions *safe_rej* and *univ_rej* remain the same, except that we have to replace *sel* by *rej*, $\tau \in \mathit{nod}(t)^n$ by $\tau \in \mathit{nod}_\bullet(t)^n$. Furthermore, $\mathcal{H}_{\mathit{sel}}$ is replaced by $\mathcal{H}_{\mathit{rej}} = \mathcal{H}_{\Sigma\times 2^{\mathcal{V}_n}}$, as safe states for rejection consider partial tuples. Hence continuations can still contain variables in their labels, and we cannot restrict the hedges to be traversed to $\mathcal{H}_{\Sigma\times\{\emptyset\}}$:

$$\mathit{safe\_rej}^A_{(\tau,\eta)}(t) = \{q \mid (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}(\eta) = q) \Rightarrow (\tau,\eta) \in \mathit{rej}_{Q_A}(t)\}$$

$$\mathit{univ\_rej}^A((a,v),\gamma,P) = \{q_0 \mid \forall h \in \mathcal{H}_{\mathit{rej}}.\ \mathit{ev\_cl}^A(h,q_0,(a,v),\gamma) \cap P \neq \emptyset\}$$

Propagation rules defining S_{rej} are also easily adapted from those defining S_{sel}.

Rule 1 Rejection states at the root are precisely non-final states:

$$S_{\mathit{rej}}((\mathit{cl},\epsilon)) = \mathit{stat}^A_{\mathrm{e}} - \mathit{fin}^A$$

Rule 2 The critical rule

$$S_{\mathit{rej}}((\mathit{op},\pi)) = \mathit{univ\_rej}^A((a,v),\gamma,S_{\mathit{rej}}((\mathit{cl},\pi)))$$

remains correct when imposing determinism and completeness on A, since this ensures that a hedge will fail iff a run on this hedge leads to a rejection state. The additional quantification over hedges in $\mathcal{H}_{\mathit{rej}}$ (in the definition of *univ_rej*), which may turn continuations into non-canonically annotated trees, makes no difficulty, since such trees cannot be recognized by A, when assuming that the language of A is canonical (it defines a query), as we do.

Rule 3 The third rule is the direct adaptation:

$$S_{\mathit{rej}}((\mathit{cl},\pi\cdot i)) = S_{\mathit{rej}}((\mathit{op},\pi))$$

Building $\boldsymbol{E}(A)$

Now the propagation rules allow to infer both $\mathit{safe\_sel}^A_{(\tau,\eta)}(t)$ and $\mathit{safe\_rej}^A_{(\tau,\eta)}(t)$ for all events η. We can see in Figure 5.2 that the definition of safe states is incompatible with a streaming evaluation. Nevertheless, the computation of safe states can be done by running the STA $E(A)$ defined in Figure 5.3. This STA does all the computation when opening nodes. In particular, when reading (op,π) it computes the safe states for the events (cl,π) and assigns them to the node state of

$$\frac{q_0 \xrightarrow{op\ (a,v):\gamma_1} q_1 \in rul^A \qquad \begin{array}{rcl}\mathcal{S}_1 & = & univ\_sel^A((a,v),\gamma_1,\mathcal{S}_0)\\ \mathcal{R}_1 & = & univ\_rej^A((a,v),\gamma_1,\mathcal{R}_0)\end{array}}{(q_0,\mathcal{S}_0,\mathcal{R}_0) \xrightarrow{op\ (a,v):(\gamma_1,\mathcal{S}_0,\mathcal{R}_0)} (q_1,\mathcal{S}_1,\mathcal{R}_1) \in rul^{E(A)}}$$

$$\frac{q_0 \xrightarrow{cl\ (a,v):\gamma_0} q_1 \in rul^A \qquad \mathcal{S}_0,\mathcal{S}_1,\mathcal{R}_0,\mathcal{R}_1 \subseteq stat_e^A}{(q_0,\mathcal{S}_0,\mathcal{R}_0) \xrightarrow{cl\ (a,v):(\gamma_0,\mathcal{S}_1,\mathcal{R}_1)} (q_1,\mathcal{S}_1,\mathcal{R}_1) \in rul^{E(A)}}$$

$$init^{E(A)} = (init^A,\ fin^A,\ stat_e^A - fin^A)$$
$$fin^{E(A)} = \{(q,\ fin^A,\ stat_e^A - fin^A) \mid q \in fin^A\}$$

Figure 5.3: Construction of $E(A)$ from A.

π (i.e. they are pushed on the stack), so that they can be used at closing. Safe states are also propagated among siblings through node states. Note that for sake of clarity, this construction does not hold for earliest selection of $()$ at the *start* event, for Boolean queries. However, this case can be processed easily by considering every possible label of the root. The signature of $E(A)$ is still $\Sigma \times 2^{\mathcal{V}_n}$, as for A. The state sets may be exponentially large, since $stat_e^{E(A)} = stat_e^A \times 2^{stat_e^A} \times 2^{stat_e^A}$ and $stat_n^{E(A)} = stat_n^A \times 2^{stat_e^A} \times 2^{stat_e^A}$. Note that E preserves determinism.

Proposition 19. *Let A be a dSTA on $\Sigma \times 2^{\mathcal{V}_n}$ that defines a query. Then $E(A)$ is a dSTA that accepts the same language as A.*

Furthermore, if r^A (resp. $r^{E(A)}$) is the unique run of A (resp. $E(A)$) on $t * \tau \in \mathcal{T}_{\Sigma\times 2^{\mathcal{V}_n}}$ then for all $\eta \in eve(\eta) - \{start\}$:

$$r_e^{E(A)}(\eta) = (r_e^A(\eta), safe\_sel^A_{(\tau,\eta)}(t), safe\_rej^A_{(\tau,\eta)}(t))$$

Proof. We prove this proposition by Lemmas 6 and 7. For the whole section, we fix A, a dSTA on $\Sigma \times 2^{\mathcal{V}_n}$ that defines a query, $t * \tau \in \mathcal{T}_{\Sigma\times 2^{\mathcal{V}_n}}$, and we suppose that r^A is the unique run of A on $t * \tau$.

We first prove that the propagation rules define the safe states. Let us consider the function f that associates a pair $(\mathcal{S},\mathcal{R}) \in 2^{stat_e^A} \times 2^{stat_e^A}$ with each event of $t*\tau$ (except *start*) using the following inference rules:

$$f((cl,\epsilon)) = (fin^A, stat_e^A - fin^A) \tag{5.1}$$

$$\frac{\pi \in nod(t) \qquad f((cl,\pi)) = (\mathcal{S},\mathcal{R}) \qquad (a,v) = lab^t(\pi) \qquad \gamma = r_n^A(\pi)}{f((op,\pi)) = (univ\_sel^A((a,v),\gamma,\mathcal{S}), univ\_rej^A((a,v),\gamma,\mathcal{R}))} \tag{5.2}$$

$$\frac{\pi \in nod(t) \qquad \pi\cdot i \in nod(t) \qquad f((op,\pi)) = (\mathcal{S},\mathcal{R})}{f((cl,\pi\cdot i)) = (\mathcal{S},\mathcal{R})} \tag{5.3}$$

Lemma 6. *For every event $\eta \in eve(t) - \{start\}$,*

$$f(\eta) = (\mathit{safe\_sel}^A_{(\tau,\eta)}(t), \mathit{safe\_rej}^A_{(\tau,\eta)}(t))$$

Proof. We proceed by induction on events of t (except *start*), according to a top-down, breadth-first, right-to-left traversal of t.

For (cl, ϵ), the result is trivial from rule (5.1) and the definitions of *safe_sel* and *safe_rej*.

Let $\eta = (op, \pi)$, and suppose that the property holds for (cl, π). From the application of rule (5.2), we know that $f(\eta) = (\mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S}), \mathit{univ\_rej}^A((a,v),\gamma,\mathcal{R}))$ with $f((cl,\pi)) = (\mathcal{S},\mathcal{R})$, $(a,v) = \mathit{lab}^t(\pi)$ and $\gamma = r^A_{\mathrm{n}}(\pi)$. By definition, we have: $\mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S}) = \{q \mid \forall h \in \mathcal{H}_{sel}.\ \mathit{ev\_cl}^A(h,q,(a,v),\gamma) \in \mathcal{S}\}$, and by induction hypothesis, $\mathcal{S} = \mathit{safe\_sel}^A_{(\tau,(cl,\pi))}(t)$.

We first prove that $\mathit{univ\_sel}^A((a,v)),\gamma,\mathcal{S}) = \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$. Suppose that $q \in \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$. Let $h \in \mathcal{H}_{sel}$, and $q' = \mathit{ev\_cl}^A(h,q,(a,v),\gamma)$. Then $q' \in \mathit{safe\_sel}^A_{(\tau,(cl,\pi))}(t)$, as sufficiency remains true for events following η. Thus, $q \in \mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S})$. Conversely, if $q \in \mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S})$ then $\tau \in \mathit{dom}_\eta(t)^n$ (consider the empty continuation). So for every $t' \in \mathcal{T}_\Sigma$ such that $\mathit{equal}_\eta(t,t')$, the hedge h of children of π in t' is in $\mathcal{H}_{sel}$. Thus $\mathit{ev\_cl}^A(h,q,(a,v),\gamma) \in \mathit{safe\_sel}^A_{(\tau,(cl,\pi))}(t)$, which means that $\tau \in Q_A(t')$, so η is sufficient for selecting τ, and $q \in \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$. Finally, $\mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S}) = \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$.

Now we prove the similar result for safe states for rejection, i.e., that: $\mathit{univ\_rej}^A((a,v),\gamma,\mathcal{R}) = \mathit{safe\_rej}^A_{(\tau,\eta)}(t)$. The difference here is that we deal with partial candidates. We write $\tau^{\preceq\eta}$ for the partial tuple obtained by replacing every component strictly after η by $\bullet$. Inclusion $\mathit{safe\_rej}^A_{(\tau,\eta)}(t) \subseteq \mathit{univ\_rej}^A((a,v),\gamma,\mathcal{S})$ holds for the same reason, namely events following η remain sufficient for rejection, even for completions of $\tau^{\preceq\eta}$. Now suppose that $q \in \mathit{univ\_rej}^A((a,v),\gamma,\mathcal{S})$. Fix $t' \in \mathcal{T}_\Sigma$ such that $\mathit{equal}_\eta(t,t')$, and let h be the hedge of children of π in t'. Then $\mathit{ev\_cl}^A(h,q,(a,v),\gamma) \in \mathit{safe\_rej}^A_{(\tau,(cl,\pi))}(t)$, and thus every completion τ' of $\tau^{\preceq\eta}$ after η fails. Hence η is sufficient for rejecting $\tau^{\preceq\eta}$, and $q \in \mathit{safe\_rej}^A_{(\tau,\eta)}(t)$.

Finally we consider $\eta = (cl, \pi\cdot i)$, and assume that the property holds for (op,π) and (cl,π). From Rule (5.3) and induction hypothesis, we obtain that: $f((cl,\pi\cdot i)) = (\mathit{safe\_sel}^A_{(\tau,(op,\pi))}(t), \mathit{safe\_rej}^A_{(\tau,(op,\pi))}(t))$.

First we prove that $\mathit{safe\_sel}^A_{(\tau,(op,\pi))}(t) = \mathit{safe\_sel}^A_{(\tau,\eta)}(t)$. We have:

$$
\begin{array}{ll}
& q \in \mathit{safe\_sel}^A_{(\tau,(op,\pi))}(t) \\
\Leftrightarrow & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((op,\pi)) = q) \Rightarrow (\tau,(op,\pi)) \in \mathit{sel}_{Q_A}(t) \\
\Leftrightarrow & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((op,\pi)) = q) \Rightarrow \\
& \quad \forall h \in \mathcal{H}_{sel}.\ \mathit{ev\_cl}^A(h,q,(a,v),\gamma) \in \mathit{safe\_sel}^A_{(\tau,(cl,\pi))}(t) \\
\overset{(i)}{\Leftrightarrow} & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((cl,\pi\cdot i)) = q) \Rightarrow (\tau,(cl,\pi\cdot i)) \in \mathit{sel}_{Q_A}(t) \\
\Leftrightarrow & q \in \mathit{safe\_sel}^A_{(\tau,(cl,\pi i))}(t)
\end{array}
$$

The equivalence (i) holds because when applying *op*-rules, STAs do not distinguish between downward or rightward moves, i.e., they do not know whether the last action was *op* or *cl*. We now show that $\mathit{safe\_rej}^A_{(\tau,(op,\pi))}(t) = \mathit{safe\_rej}^A_{(\tau,\eta)}(t)$:

$$
\begin{array}{ll}
& q \in \mathit{safe\_rej}^A_{(\tau,(op,\pi))}(t) \\
\Leftrightarrow & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((op,\pi)) = q) \Rightarrow (\tau,(op,\pi)) \in \mathit{rej}_{Q_A}(t) \\
\Leftrightarrow & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((op,\pi)) = q) \Rightarrow \\
& \quad \forall h \in \mathcal{H}_{rej}.\ \mathit{ev\_cl}^A(h,q,(a,v),\gamma) \in \mathit{safe\_rej}^A_{(\tau_h,(cl,\pi))}(t) \\
\Leftrightarrow & (\exists r \in \mathit{p\_runs}^A(t * \tau) \wedge r_{\mathrm{e}}((cl,\pi\cdot i)) = q) \Rightarrow (\tau,(cl,\pi\cdot i)) \in \mathit{rej}_{Q_A}(t) \\
\Leftrightarrow & q \in \mathit{safe\_rej}^A_{(\tau,(cl,\pi i))}(t)
\end{array}
$$

where τ_h is obtained from τ by adding variables in h. □

Lemma 7. *There is a run* $(r_e^{E(A)}, r_n^{E(A)})$ *of* $E(A)$ *on* $t * \tau \in L(A)$*, and for every event* $\eta \in \mathit{eve}(t) - \{\mathit{start}\}$*,*

$$r_e^{E(A)}(\eta) = (r_e^A(\eta), \mathcal{S}, \mathcal{R}) \text{ with } (\mathcal{S},\mathcal{R}) = f(\eta)$$

Proof. Inference schemas defining $E(A)$ show that every run r of A has a unique corresponding run r' in $E(A)$, and r is the first component of r'. Again, we use an induction on events of t (except *start*) according to a top-down, breadth-first, left-to-right traversal of t.

For $\eta = (cl,\epsilon)$, we have $f(\eta) = (\mathit{fin}^A, \mathit{stat}^A_{\mathrm{e}} - \mathit{fin}^A)$. At the root, we have $r_{\mathrm{n}}(\epsilon) = (r^A_{\mathrm{n}}(\epsilon), \mathit{fin}^A, \mathit{stat}^A_{\mathrm{e}} - \mathit{fin}^A)$, so $r_{\mathrm{e}}((cl,\epsilon)) = (r^A_{\mathrm{e}}((cl,\epsilon)), \mathit{fin}^A, \mathit{stat}^A_{\mathrm{e}} - \mathit{fin}^A)$.

Now consider that $\eta = (op,\pi)$ and suppose that we have $r^{E(A)}_{\mathrm{e}}((cl,\pi)) = (r^A_{\mathrm{e}}((cl,\pi)), \mathcal{S}', \mathcal{R}')$ with $(\mathcal{S}',\mathcal{R}') = f((cl,\pi))$. This implies that $r^{E(A)}_{\mathrm{n}}(\pi) = (r^A_{\mathrm{n}}(\pi), \mathcal{S}', \mathcal{R}')$, so we get $\mathcal{S}' = \mathit{univ\_sel}^A((a,v),\gamma,\mathcal{S})$, $\mathcal{R}' = \mathit{univ\_rej}^A((a,v),\gamma,\mathcal{R})$ and $r^{E(A)}_{\mathrm{e}}(\eta) = (r^A_{\mathrm{e}}(\eta), \mathcal{S}, \mathcal{R})$ where $(a,v) = \mathit{lab}^t(\pi)$ and $\gamma = r^A_{\mathrm{n}}(\pi)$. Hence, $(\mathcal{S},\mathcal{R}) = f((op,\pi))$.

Finally, let us assume that $\eta = (cl, \pi\cdot i)$ and also that $r^{E(A)}_{\mathrm{e}}((op,\pi)) = (r^A_{\mathrm{e}}((op,\pi)), \mathcal{S}, \mathcal{R})$ with $\mathcal{S},\mathcal{R}$ defined by $(\mathcal{S},\mathcal{R}) = f((op,\pi))$. By an immediate induction on children of π, each child $\pi{\cdot}j$ of π verifies $r^{E(A)}_{\mathrm{n}}(\pi{\cdot}j) = (r^A_{\mathrm{n}}(\pi{\cdot}j), \mathcal{S}, \mathcal{R})$ and for the state $r^{E(A)}_{\mathrm{e}}((cl,\pi{\cdot}j)) = (r^A_{\mathrm{e}}((cl,\pi{\cdot}j)), \mathcal{S}, \mathcal{R})$, and in particular for $j = i$. From rule (5.3) of the definition of f, we know that $(\mathcal{S},\mathcal{R}) = f((cl,\pi\cdot i))$. □

These two lemmas finally prove the correctness of $E(A)$. □

$$\frac{(a,v) \in \Sigma \times \{\emptyset\} \qquad q_1 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_3 \in \mathit{rul}^A \qquad q_4 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^A}{\mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q_1,q_2) \text{ :- } \mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q_3,q_4).}$$

$$\frac{q \in \mathit{stat}^A_{\mathrm{e}}}{\mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q,q).} \qquad \frac{q_1,q_2,q_3 \in \mathit{stat}^A_{\mathrm{e}}}{\mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q_1,q_2) \text{ :- } \mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q_1,q_3), \mathit{acc}_{\mathcal{H}_{\mathit{sel}}}(q_3,q_2).}$$

Figure 5.4: Inference rules for the definition of $\mathit{acc}^A_{\mathcal{H}_{\mathit{sel}}}$.

Running automaton $E(A)$ for a candidate permits to test sufficiency for selection and rejection at the event when it happens. At most one run has to be processed per candidate, thanks to determinism.

5.4.2 Generic EQA Algorithm and its Instantiation for dSTAs

We present an EQA algorithm for queries defined by dSTAs A which runs in polynomial time per step and candidate. The idea is to run the earliest automaton $E(A)$ of Section 5.4.1 on the input stream in order to decide selection and rejection sufficiency for all answer candidates at all time points, without constructing $E(A)$ explicitly.

Running $E(A)$ on the fly

Given a dSTA A over $\Sigma \times 2^{\mathcal{V}_n}$ and a tree $t * \tau$ over the same signature, we want to compute a run of $E(A)$ on $t*\tau$ in polynomial time in the size of A. The application of closing rules of $E(A)$ is easy, since it only has to look for a rule of A. Applying opening rules of $E(A)$ is a little more tedious, since we have to compute the sets $\mathit{univ\_sel}((a,v),\gamma,P)$ and $\mathit{univ\_rej}((a,v),\gamma,P')$ while given $a \in \Sigma$, $\gamma \in \mathit{stat}^A_{\mathrm{n}}$, and $P, P' \subseteq \mathit{stat}^A_{\mathrm{e}}$.

When assuming the completeness of A in addition to determinism (which can be ensured in polynomial time for a fixed arity n), these sets can be computed by reduction to information on accessibility through hedges for A. Given a set $H \subseteq \mathcal{H}_{\Sigma\times 2^{\mathcal{V}_n}}$ of hedges, and event states $q_1, q_2 \in \mathit{stat}^A_{\mathrm{e}}$, we define the following accessibility predicate:

$$\mathit{acc}^A_H(q_1,q_2) \quad \Leftrightarrow \quad \exists h \in H.\ \mathit{runs}^A_{q_1 \to q_2}(h) \neq \emptyset$$

We compute it for $\mathcal{H}_{\mathit{sel}} = \mathcal{H}_{\Sigma\times\{\emptyset\}}$ and $\mathcal{H}_{\mathit{rej}} = \mathcal{H}_{\Sigma\times 2^{\mathcal{V}_n}}$, with the Datalog program in Figures 5.4 and 5.5.

$$\frac{(a,v) \in \Sigma \times 2^{\mathcal{V}_n} \qquad q_1 \xrightarrow{op\ (a,v):\gamma} q_3 \in \mathit{rul}^A \qquad q_4 \xrightarrow{cl\ (a,v):\gamma} q_2 \in \mathit{rul}^A}{\mathit{acc}_{\mathcal{H}_{rej}}(q_1,q_2)\ \text{:-}\ \mathit{acc}_{\mathcal{H}_{rej}}(q_3,q_4).}$$

$$\frac{q \in \mathit{stat}_{\mathrm{e}}^A}{\mathit{acc}_{\mathcal{H}_{rej}}(q,q).} \qquad \frac{q_1,q_2,q_3 \in \mathit{stat}_{\mathrm{e}}^A}{\mathit{acc}_{\mathcal{H}_{rej}}(q_1,q_2)\ \text{:-}\ \mathit{acc}_{\mathcal{H}_{rej}}(q_1,q_3),\mathit{acc}_{\mathcal{H}_{rej}}(q_3,q_2).}$$

Figure 5.5: Inference rules for the definition of $\mathit{acc}^A_{\mathcal{H}_{rej}}$.

Proposition 20. *The collections of values $\mathit{acc}^A_{\mathcal{H}_{sel}}(q_1,q_2)$ and $\mathit{acc}^A_{\mathcal{H}_{rej}}(q_1,q_2)$ can be computed in time $O(|\mathit{rul}^A|^2 + |\mathit{stat}_e^A|^3)$ for every complete dSTA A.*

To explain the computation of $\mathit{univ\_sel}^A$, we introduce $\mathit{beforeClose}^A((a,v),\gamma,P)$, the set of states that lead to a state of P after closing (a,v) with γ:

$$\mathit{beforeClose}^A((a,v),\gamma,P) \;=\; \{q_0 \mid \exists q_1 \in P.\ q_0 \xrightarrow{cl\ (a,v):\gamma} q_1 \in \mathit{rul}^A\}$$

Lemma 8. *For deterministic and complete A, and for $X \in \{sel, rej\}$, the safe states $\mathit{univ\_X}^A((a,v),\gamma,P)$ are equal to:*

$$\{q \mid \forall q_0.\ \mathit{acc}^A_{H_X}(q,q_0) \Rightarrow q_0 \in \mathit{beforeClose}^A((a,v),\gamma,P)\}$$

Proof. Immediate from the definitions. □

We will see in the sequel how the relations $\mathit{acc}_{\mathcal{H}_{sel}}$ and $\mathit{acc}_{\mathcal{H}_{rej}}$ are precomputed and then reused dynamically.

Generic Algorithm

Our algorithm will be obtained by instantiating the skeleton in Figure 5.6 of a generic EQA algorithm, which is parameterized by a class $\mathcal{Q}$ of query definitions. In our computational model, such an algorithm, for a given query Q, is implemented by an SRAM, where candidates are stored in the working memory, whereas the node identifiers are stored in registers. The static input of the algorithm is a query definition $e \in \mathcal{Q}$, and its dynamic input on the stream is its ordered set of events. We assume that the stream is already parsed, as in our SRAM model. Our algorithm adds the tuples of $Q(t)$ to the external output collection incrementally at the earliest possible event. The main idea is to generate all candidate tuples, test their aliveness repeatedly, output selected candidates and remove rejected candidates.

```
fun answer(e,t) % e ∈ Q, t ∈ dom(Q)
   let candidates = set.new(∅)
in
   for η in eve(t) in streaming−order do
      candidates.update(η)
      for τ in candidates do
        if (τ,η) ∈ sel_{Q_e}(t)
           then add−output(τ)
                candidates.remove(τ)
        elseif (τ,η) ∈ rej_{Q_e}(t)
           then candidates.remove(τ)
```

Figure 5.6: Generic EQA algorithm for a class $\mathcal{Q}$ of query definitions.

Instantiation for dSTAs

Now suppose that the query is defined by a dSTA A. For every candidate τ we maintain its configuration in $E(A)$, i.e. its current state $(q, \mathcal{S}, \mathcal{R}) \in \mathit{stat}_{\mathrm{e}}^{E(A)}$ and a sequence $\Upsilon \in (\mathit{stat}_{\mathrm{n}}^{E(A)})^*$ inside a stack. Sufficiency for selection $(\tau, e) \in \mathit{sel}_{Q_A}(t)$ is verified by testing $q \in \mathcal{S}$, and sufficiency for rejection $(\tau, e) \in \mathit{rej}_{Q_A}(t)$ by checking $q \in \mathcal{R}$. Updating the current state is done by applying a rule of $E(A)$, that we can compute using the alternative definition of *univ_X* in Lemma 8.

Updating the current set of candidates at event η means to apply a rule of $E(A)$ to the current state $(q, \mathcal{S}, \mathcal{R}) \in E(A)$, and for opening events to create all new candidates, where the current node is used. Let C the number of candidates to be processed at event (op, π). Each of the C candidates originates from an alive candidate at the previous event $\mathit{pr}((\mathit{op}, \pi))$, with a possible completion of $\bullet$-components with π. We distinguish between candidates that get safe for selection or rejection at (op, π) from those that are still alive. We write $i = \mathit{simult\_safe}_{Q_A}(t)$ for a bound on the former (when iterating on $\mathit{eve}(t)$), while the second is bounded by the concurrency $c = \mathit{concur}_{Q_A}(t)$. Hence we have $C \leq c + i$. Let us formalize $\mathit{simult\_safe}_Q(t)$, the maximal number of candidates becoming safe for selection or rejection at the same event. For a tuple τ and a node π, we write $\tau{-}\pi$ for the tuple obtained from τ by replacing π by $\bullet$.

$$\mathit{simult\_safe}_Q(t) = \max_{\pi \in \mathit{nod}(t)} \left| \left\{ \tau \;\middle|\; \begin{array}{r} \tau{-}\pi \text{ is alive at event } \mathit{pr}((\mathit{op}, \pi)) \\ \wedge \quad \tau \text{ is not alive at event } (\mathit{op}, \pi) \end{array} \right\} \right|$$

$$= \max_{\pi \in \mathit{nod}(t)} \left| \left\{ \tau \;\middle|\; \begin{array}{r} (\tau{-}\pi,\ \mathit{pr}((\mathit{op}, \pi))) \notin \mathit{sel}_Q(t) \cup \mathit{rej}_Q(t) \\ \wedge\ (\tau,\ (\mathit{op}, \pi)) \in \mathit{sel}_Q(t) \cup \mathit{rej}_Q(t) \end{array} \right\} \right|$$

The maximal value for $\mathit{simult\_safe}_Q(t)$ is reached when there are many alive candidates $\tau{-}\pi$ at $\mathit{pr}((\mathit{op}, \pi))$, and all the candidates τ are not alive at (op, π). There can be at most 2^n values for τ, for a given $\tau{-}\pi$, so we get the following upper

bound:

$$\mathit{simult\_safe}_Q(t) \leq 2^n \cdot \mathit{concur}_Q(t)$$

We have already seen how to apply rules of $E(A)$ in polynomial time in the size of A. The node state of the rule is pushed to stack Υ for opening events, and popped from Υ for closing events.

Theorem 4. *For every complete dSTA A recognizing a canonical language over $\Sigma \times 2^{\mathcal{V}_n}$, one can compute in time $O(|A|^3)$ an SRAM $\mathcal{M}_A$ computing the query Q_A and using at each event:*

- $\mathit{Time}(\mathcal{M}_A, t) = O((c+i) \cdot |A|^2)$
- $\mathit{Space}(\mathcal{M}_A, t) = O(c \cdot d \cdot |A|)$

with $c = \mathit{concur}_{Q_A}(t)$, $i = \mathit{simult\_safe}_{Q_A}(t)$, and $d = \mathit{depth}(t)$.

Proof. The computation of $\mathcal{M}_A$ from A consists mainly in building the accessibility relations $\mathit{acc}^A_{H_X}$ for $X \in \{\mathit{sel}, \mathit{rej}\}$. We can compute these relations for A in time $O(|A|^3)$ according to Proposition 20. These relations are stored in the finite state control.

Processing an opening event requires more computations than a closing one, as it needs to determine the sufficient events. Given a label $a \in \Sigma$ and a current state $(q_0, \mathcal{S}_0, \mathcal{R}_0)$ for the partial run of the candidate, we have to consider the rules of A of the form $q_0 \xrightarrow{\mathit{op}\ (a,v):\gamma_1} q_1$. For each of these rules, the computation of $\mathit{beforeClose}((a,v), \gamma_1, \mathcal{S}_0)$ can be performed in time $O(|\mathit{rul}^A|)$. Then, the computation of $\mathit{univ\_X}$ where $X \in \{\mathit{sel}, \mathit{rej}\}$ can be done in time $O(|\mathit{stat}^A_{\mathrm{e}}|^2)$, by Lemma 8. There are at most $(c+i)$ such updates to process per event.

The fact that this algorithm is an EQA algorithm implies that at most c candidates are stored at a time. For each candidate, we have to store the node states of its ancestors and its current event state, which requires $d \cdot |A|$. □

5.4.3 Adding Schemas

With respect to sufficiency checking, we can integrate the schema into the query. Validation of the document with respect to the schema is an independent task, that we run in parallel. Given an n-ary query Q with a schema $\mathit{dom}(Q) \subseteq \mathcal{T}_\Sigma$, we define the queries Q_{sel} and Q_{rej} with universal schema:

$$Q_{\mathit{sel}}(t) = \begin{cases} Q(t) & \text{if } t \in \mathit{dom}(Q) \\ \mathit{nod}(t)^n & \text{otherwise} \end{cases} \qquad \mathit{dom}(Q_{\mathit{sel}}) = \mathcal{T}_\Sigma$$

$$Q_{\mathit{rej}}(t) = \begin{cases} Q(t) & \text{if } t \in \mathit{dom}(Q) \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathit{dom}(Q_{\mathit{rej}}) = \mathcal{T}_\Sigma$$

$$\frac{q_0 \xrightarrow{op\ (a,v):\gamma_1} q_1 \in rul^A \qquad q'_0 \xrightarrow{op\ a:\gamma'_1} q'_1 \in rul^B}{(q_0,q'_0) \xrightarrow{op\ (a,v):(\gamma_1,\gamma'_1)} (q_1,q'_1)\ \in rul^{A_{sel}}}$$

$$\frac{q_0 \xrightarrow{cl\ (a,v):\gamma_0} q_1 \in rul^A \qquad q'_0 \xrightarrow{cl\ a:\gamma'_0} q'_1 \in rul^B}{(q_0,q'_0) \xrightarrow{cl\ (a,v):(\gamma_0,\gamma'_0)} (q_1,q'_1)\ \in rul^{A_{sel}}}$$

$$init^{A_{sel}} = init^A \times init^B \qquad fin^{A_{sel}} = (fin^A \times fin^B)\ \cup\ (stat^A_e \times (stat^B_e - fin^B))$$

Figure 5.7: Construction of A_{sel} from A and B.

Lemma 9. *$sel_Q = sel_{Q_{sel}}$ and $rej_Q = rej_{Q_{rej}}$.*

Proof. Straightforward from definitions.

$$\begin{array}{lll}
(\tau,\eta) \in sel_{Q_{sel}} & \text{iff} & \tau \in dom_\eta(t)^n\ \wedge\ \forall t' \in \mathcal{T}_\Sigma.\ equal_\eta(t,t') \Rightarrow \tau \in Q_{sel}(t') \\
& \text{iff} & \tau \in dom_\eta(t)^n\ \wedge\ \forall t' \in dom(Q).\ equal_\eta(t,t') \Rightarrow \tau \in Q(t') \\
(\tau,\eta) \in rej_{Q_{rej}} & \text{iff} & \left\{ \begin{array}{l} \tau \in dom^\bullet_\eta(t)^n\ \wedge \\ \forall t' \in \mathcal{T}_\Sigma.\ equal_\eta(t,t') \Rightarrow \\ \qquad \forall \tau' \in compl(\tau,t',\eta).\ \tau' \notin Q_{rej}(t') \end{array} \right. \\
& \text{iff} & \left\{ \begin{array}{l} \tau \in dom^\bullet_\eta(t)^n\ \wedge \\ \forall t' \in dom(Q).\ equal_\eta(t,t') \Rightarrow \\ \qquad \forall \tau' \in compl(\tau,t',\eta).\ \tau' \notin Q(t') \end{array} \right.
\end{array}$$

□

For selection detection, the idea is to build an automaton A_{sel} recognizing Q_{sel} from the STAs A and B recognizing $Q_{A,B}$. This automaton will be similar to the product automaton of A and B, but final states will be enriched by all invalid selections, as introduced in the definition of Q_{sel}. Figure 5.7 shows how to obtain the STA A_{sel}. Prior to this construction, A and B must be determinized and completed. For rejection detection, we proceed the same way to obtain A_{rej} such that $Q_{A_{rej}} = Q_{rej}$. The only difference between A_{sel} and A_{rej} lies in the final states: $fin^{A_{rej}} = fin^A \times fin^B$.

Lemma 10. *$L(A_{sel}) = L_{Q_{sel}}$ and $L(A_{rej}) = L_{Q_{rej}}$.*

This way, we can compute the safe states for selection with $E(A_{sel})$ and the safe states for rejection with $E(A_{rej})$. From an implementation point of view, there is no need to compute the safe states for rejection of $E(A_{sel})$ and the safe states for selection of $E(A_{rej})$. Thus, we can run the efficient algorithm presented in Section 5.4.2 and compute the same amount of safe states as for $E(A)$, but on a bigger automaton. We get the following result for our EQA algorithm with schemas.

Theorem 5. *For every complete dSTA A recognizing a canonical language over $\Sigma \times 2^{\mathcal{V}_n}$ and every complete dSTA B, one can compute in time $O(|A|^3 \cdot |B|^3)$ an SRAM $\mathcal{M}_{A,B}$ computing the query $Q_{A,B}$, where $\mathcal{M}_{A,B}$ uses for each event:*

- *Time*$(\mathcal{M}_{A,B}, t) = O((c+i) \cdot |A|^2 \cdot |B|^2)$
- *Space*$(\mathcal{M}_{A,B}, t) = O(c \cdot d \cdot |A| \cdot |B|)$

with $c =$ *concur*$_{Q_{A,B}}(t)$, $i =$ *simult_safe*$_{Q_{A,B}}(t)$, *and* $d =$ *depth*(t).

Proof. The complexity analysis is similar to Theorem 4. The difference is that we use A_{sel} and A_{rej} instead of A, and $|A_{sel}|$ and $|A_{rej}|$ are in $O(|A| \cdot |B|)$, and can be computed with this time complexity. □

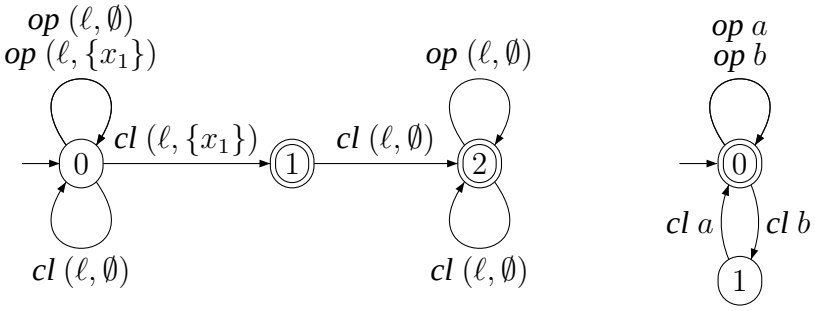
5.4.4 Example Run of the Algorithm with Schema

For illustration, let us consider the monadic query Q_0 that selects all nodes without next sibling. It can be defined in MSO by the formula $\neg\exists y.\ ns(x, y)$. The root of t is selected, and this can be decided when opening it. Without schema, membership $\pi \in Q_0(t)$ cannot always be decided at opening time, so the algorithm needs to memorize nodes until, either encountering the opening event of the next sibling (for nodes $\pi \notin Q_0(t)$) or the closing event of the father (for selected nodes $\pi \in Q_0(t)$). When assuming the DTD $a \to (a^*b)^*$ and $b \to \epsilon$, one knows that all a-nodes except the root have a next sibling in all trees satisfying the DTD, so selection of a nodes be decided early at opening time. For b-nodes, selection can still be decided only later, when closing the parent. We consider the schema S_0 which corresponds to the DTD $\{a \to a^*b,\ b \to \epsilon\}$, and choose it as domain of Q_0: *dom*$(Q_0) = S_0$. We show how the algorithm would behave on this input.

For clarity, we omit node states in the following figures, as only one occurs in each automaton. Moreover, whenever ℓ occurs in a rule, this means that this rules exists for $\ell \in \{a, b\}$. Let A be the dSTA represented in Figure 5.8(a), and B the dSTA in Figure 5.8(b). We have $Q_0 = Q_{A,B}$.

We start by completing A with the sink state 3 and B with the sink state 2. By applying the inference rules in Figure 5.7, we obtain the STA A_{sel} represented in Figure 5.9 (states resulting from completion are omitted for clarity). The STA A_{rej} only differs on final states.

Then we compute the relations *acc*$_{\mathcal{H}_{sel}}$ and *acc*$_{\mathcal{H}_{rej}}$. Figure 5.10 is an array of Booleans representing the relation *acc*$_{\mathcal{H}_{rej}}$. States (q_0, q_1) are written q_0q_1 for sake of conciseness. The relation *acc*$_{\mathcal{H}_{sel}}$ is obtained from this array by replacing values in italics by 0. For instance, *acc*$_{\mathcal{H}_{rej}}((0,2),(1,2))$ holds, but not *acc*$_{\mathcal{H}_{sel}}((0,2),(1,2))$.



(a) dSTA A recognizing L_{Q_0}.

(b) dSTA B recognizing $L(B) = \mathit{dom}(Q_0)$.

Figure 5.8: Input dSTAs.

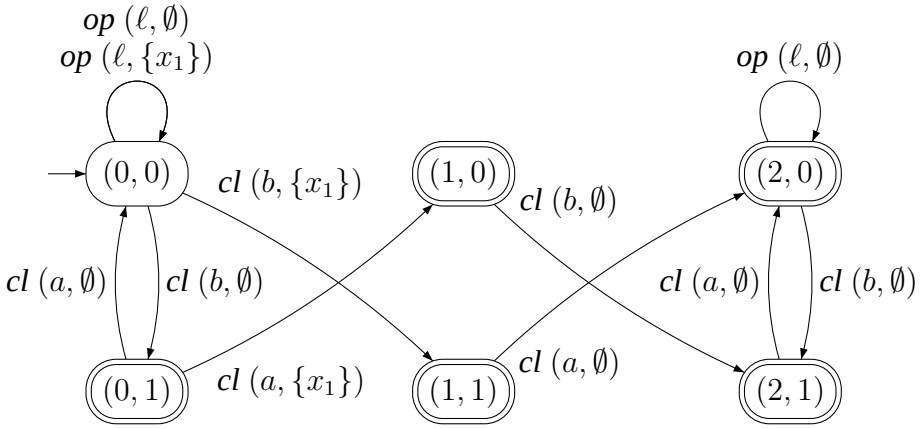


Figure 5.9: The dSTA A_{sel} obtained from A and B (sink states are omitted).

Suppose that we want to compute the safe states at a root labeled by $(a, \emptyset)$ on our example. This corresponds to computing $\mathit{safe\_sel}^{A_{sel}}((a, \emptyset), \gamma, \mathit{fin}^{A_{sel}})$, where γ is the only node state in A_{sel}. First, we obtain from the "*cl*" rules of A_{sel}:

$$\mathit{beforeClose}^{A_{sel}}((a, \emptyset), \gamma, \mathit{fin}^{A_{sel}}) = \{(0,0), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,2)\}$$

We denote this set BC_1. From the previous section, we can look at which states q verify $\forall q_0.\ \mathit{acc}_{\mathcal{H}_{sel}}(q, q_0) \Rightarrow q_0 \in BC_1$. These states are the safe states:

$$\mathit{safe\_sel}^{A_{sel}}((a, \emptyset), \gamma, \mathit{fin}^{A_{sel}}) = \{(0,2), (1,1), (1,2), (2,0), (2,1), (2,2), (3,2)\}$$

Using this processing at each opening event for safe states for selection and rejection, we obtain the run on the canonical tree represented in Figure 5.11. Here, safe

$acc_{\mathcal{H}_{rej}}$	00	01	02	10	11	12	20	21	22	30	31	32
00	1	1	1	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
01	0	1	1	0	0	*1*	0	0	*1*	0	0	*1*
02	0	0	1	0	0	*1*	0	0	*1*	0	0	*1*
10	0	0	0	1	0	0	0	0	0	1	1	1
11	0	0	0	0	1	0	0	0	0	0	0	1
12	0	0	0	0	0	1	0	0	0	0	0	1
20	0	0	0	0	0	0	1	1	1	*1*	*1*	*1*
21	0	0	0	0	0	0	0	1	1	0	0	*1*
22	0	0	0	0	0	0	0	0	1	0	0	*1*
30	0	0	0	0	0	0	0	0	0	1	1	1
31	0	0	0	0	0	0	0	0	0	0	1	1
32	0	0	0	0	0	0	0	0	0	0	0	1

Figure 5.10: $acc_{\mathcal{H}_{rej}}$ associated to Q_0 and S_0.

states for selection $\mathcal{S}$ are those provided by A_{sel} and safe states for rejection $\mathcal{R}$ are those provided by A_{rej}. We only represent them as they are the only relevant ones (safe states for rejection computed by A_{sel} are useless, for instance).

5.4.5 Implementation

We are currently implementing the algorithm described above, in a project named EvoXs [GP09]. A first step is to have an earliest query answering algorithm for queries defined by dSTAs. Then we would like to implement the translation of XPath fragments to dSTAs, in order to have an EQA XPath evaluator. The translation of XPath fragments to dSTAs is provided in Chapter 6.

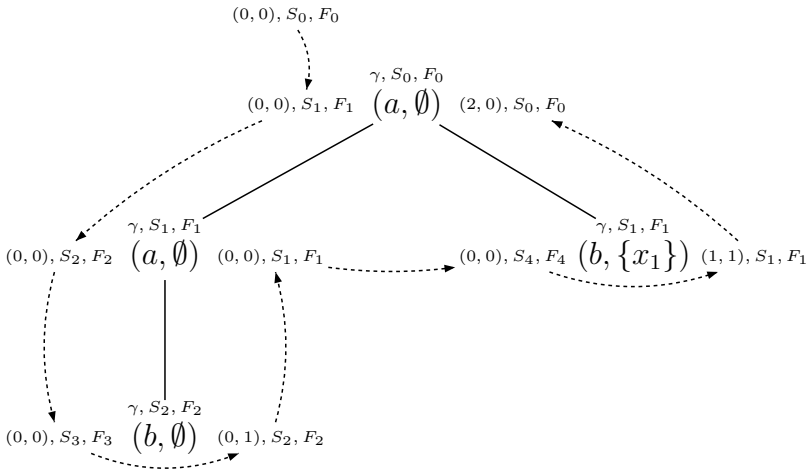
We provide here a more precise and efficient procedure for the computation of safe states *univ_X* where $X \in \{sel, rej\}$ for a dSTA A. We first exhibit some properties of the function mapping sets P to *beforeClose*$((a, v), \gamma, P)$, where (a, v) and γ are fixed.

Lemma 11. *For every* $(a, v) \in \Sigma \times 2^{\mathcal{V}_n}$, $\gamma \in stat_n^A$, *and* $P_1, P_2 \subseteq stat_e^A$*:*

$$beforeClose((a, v), \gamma, P_1 \cup P_2) = beforeClose((a, v), \gamma, P_1) \cup beforeClose(a, \gamma, P_2)$$

So we get $beforeClose((a, v), \gamma, P_2) = \cup_{q \in P_2} beforeClose((a, v), \gamma, \{q\})$. Hence we can precompute *beforeClose*$((a, v), \gamma, \{q\})$ for each $a \in \Sigma$, $\gamma \in stat_{\mathrm{n}}^A$ and $q \in stat_{\mathrm{e}}^A$, and reuse it for computing *beforeClose*$((a, v), \gamma, P_2)$. This preprocessing requires time $O(|\Sigma| \cdot |A|^3)$ and space $O(|\Sigma| \cdot |A|^2)$. This could also be replaced by a computation on-demand, and by keeping in memory the results.

Now we look into more details the properties of the function mapping sets P to *univ_X*$((a, v), \gamma, P)$ for fixed (a, v) and γ.



(a) Run of the algorithm on a tree for one candidate.

S_0={(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 1), (3, 2)}
S_1=S_2={(0, 2), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 2)}
S_3={(0, 1), (0, 2), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 1), (3, 2)}
S_4={(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)}
F_0={(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
F_1={(0, 1), (0, 2), (1, 0), (1, 2), (2, 2), (3, 0), (3, 1), (3, 2)}
F_2={(0, 2), (1, 0), (1, 2), (2, 2), (3, 0), (3, 1), (3, 2)}
F_3={(0, 1), (0, 2), (1, 1), (1, 2), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}
F_4={(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)}

(b) Sets involved in this run.

Figure 5.11: Run of the algorithm on a tree.

Lemma 12. *For every* $(a, v) \in \Sigma \times 2^{\mathcal{V}_n}$, $\gamma \in \mathit{stat}_n^A$, $P_1, P_2 \subseteq \mathit{stat}_e^A$ *and* $X \in \{\mathit{sel}, \mathit{rej}\}$*:*

$$\mathit{univ\_X}((a, v), \gamma, P_1 \cup P_2) \supseteq \mathit{univ\_X}((a, v), \gamma, P_1) \cup \mathit{univ\_X}((a, v), \gamma, P_2)$$

A consequence is that the function mapping sets $P \to \mathit{univ\_X}((a, v), \gamma, P)$ is monotonic. Note that in the general case, $\mathit{univ\_X}((a, v), \gamma, P_1 \cup P_2) \not\subseteq \mathit{univ\_X}((a, v), \gamma, P_1) \cup \mathit{univ\_X}((a, v), \gamma, P_2)$. For instance, in our example, $(0, 2) \notin \mathit{univ\_rej}(a_1, 0, \{(1, 2)\})$ and $(0, 2) \notin \mathit{univ\_rej}(a_1, 0, \{(3, 2)\})$, but $(0, 2) \in \mathit{univ\_rej}(a_1, 0, \{(1, 2), (3, 2)\})$.

Algorithm in Figure 5.12 uses these results, and also the fact that, from Lemma 8, $\mathit{univ\_X}((a, v), \gamma, P_2) \subseteq \mathit{beforeClose}((a, v), \gamma, P_2)$. Note that if we choose to store all the computations of safe states (used in the first *for* loop), this can use memory of size $O(|\Sigma| \cdot |\mathit{stat}_n^A| \cdot |2^{\mathit{stat}_e^A}|^2)$. However, this can be weakened.

```
fun univ_X((a, v), γ, P)
  let safeStates = set.new(∅)
  let beforeCl = ∪_{q∈P} beforeClose((a, v), γ, {q})
  let agenda = beforeCl
in

  // first we set the agenda to what really needs to be computed
  for P_1 ⊆ P such that univ_X((a, v), γ, P_1) is memorized
    let U = univ_X((a, v), γ, P_1)
    in
      safeStates.add(U)
      agenda.remove(U)

  // then we perform the needed computations
  for q in agenda
    is_safe = true
    for q' such that acc_{H_X}(q, q')
      if q' not in beforeCl
        is_safe = false
    if is_safe
      safeStates.add(q)

  return safeStates
```

Figure 5.12: Algorithm computing *univ_X*$((a, v), \gamma, P)$.

For instance a good trade-off between memory and time consumption can be to store all safe states of all previous siblings of the current branch. The reason is that the safe states at opening $(op, \pi{\cdot}i)$ are computed from the safe states at closing $(cl, \pi \cdot i)$, which are the same for all siblings (as they are equal to the safe states at (op, π)). Thus, if two siblings have the same label and the same associated node state, their safe states are equal.

5.5 Streamability of dSTAs

The EQA algorithm previously described gives a PTIME procedure for evaluating queries defined by dSTAs, while keeping only alive candidates in memory. As a consequence, dSTAs are a streamable query class when trees are shallow, i.e. when there is a bound on the depth of valid trees. Let $\mathcal{Q}^d_{\text{dSTAs}}$ be the class of queries of fixed arity n where all expressions $e \in \mathcal{Q}$ are composed of two dSTAs A,B defining $Q_{A,B}$, with the semantic restriction that schemas $L(B)$ only contains trees of depth at most d.

Theorem 6. *For every $d \in \mathbb{N}$, the class $\mathcal{Q}^d_{dSTAs}$ is m-streamable for all $m \in \mathbb{N}_0$.*

Proof. The EQA algorithm requires complete dSTAs, so a first step is to complete A and B. This can be done in time $O(|\Sigma| \cdot 2^n \cdot |stat^A_e| \cdot |stat^A_n| + |rul^A|)$ for A, and similarly for B. As n is fixed, this is a PTIME procedure. Then the precomputation

step of the EQA algorithm is in PTIME, so we can find a polynomial p_0 for the computation of SRAMs $\mathcal{M}_{A,B}$ computing $Q_{A,B}$.

If we suppose that the concurrency of t is less than a given m, then $\mathit{concur}_{Q_{A,B}}(t) + \mathit{simult\_safe}_{Q_{A,B}}(t) \leq (2^n + 1) \cdot m$, as we know that $\mathit{simult\_safe}_{Q_{A,B}}(t) \leq 2^n \cdot \mathit{concur}_{Q_{A,B}}(t)$. Hence from Theorem 5, the time used per event is in $O((2^n+1) \cdot m \cdot |A|^2 \cdot |B|^2)$, and we can find a polynomial p_2 bounding this, as n is fixed. The space complexity is in $O(m \cdot \mathit{depth}(t) \cdot |A| \cdot |B|)$, and $\mathit{depth}(t)$ is bounded by d. Hence a polynomial p_2 bounding the space complexity exists. □

However dSTAs allow to define queries with unbounded concurrency, so they are not ∞-streamable.

Proposition 21. *For every $d \geq 2$, the class $\mathcal{Q}^d_{\mathit{dSTAs}}$ is not ∞-streamable.*

Proof. We can for instance define a dSTA A for the query selecting all children of the root, if the last one is labeled by a. For this query and any value of k, the tree t with $k+2$ children is such that $\mathit{concur\_nod}_{Q_A}(t) > k$. By Proposition 9, $\mathcal{Q}^d_{\text{dSTAs}}$ is not ∞-streamable for shallow trees containing trees of depth $d = 2$. □

5.6 Conclusion

In this chapter, we have seen that dSTAs enjoy good streamability properties, by proposing an EQA algorithm using low buffering (close to optimal) while still being in polynomial. More generally, EQA is time consuming for numerous query classes. We believe that dSTAs are the good model for efficient XML stream processing, and conjecture that a query class is m-streamable for all $m \in \mathbb{N}_0$ iff there is a PTIME translation to dSTAs. In Chapter 6, we provide such a translation for a fragment of XPath, thus proving its m-streamability for all $m \in \mathbb{N}_0$. Finding ∞-streamable classes of dSTAs-defined queries by syntactic and semantic restrictions is an open issue.

Processing XML streams often implies a tradeoff between time and space complexity. In earliest query answering algorithms, the priority is given to a minimal space consumption. In the future, we plan to validate our algorithm experimentally. For some queries, significant improvements are expected on space consumption. In this chapter we provided some details on efficient computation of safe states. Some further work is also planned, to get a concise data structure for the set of alive candidates to be buffered. Another challenge is to avoid the completion of the input dSTAs A and B, as the completeness was always assumed, but the completion requires time in $O(|\Sigma| \cdot 2^n \cdot |\mathit{stat}^A_{\text{e}}| \cdot |\mathit{stat}_{\text{n}}|^A)$ for A, and similarly

for B. It will also be interesting to distinguish which queries are efficiently processed. In Chapter 7, we give a procedure to detect some of these queries, namely those having bounded delay and concurrency.

Another future work is to investigate how the EQA algorithm can be extended. We propose three extensions in the sequel. The first possible extension is on the query class. We studied queries defined by dSTAs, but is it possible to adapt the algorithm for deterministic pushdown automata? This seems reasonable, as STAs are a reformulation of visibly pushdown automata, i.e. pushdown automata where the letter gives the action (push or pop). Without determinism, we cannot build a PTIME EQA algorithm (by Proposition 18), and determinism, together with our representation through canonical languages, were crucial in our construction. The second extension is to consider other structures, and for instance directed acyclic graphs. These structures models for instance XML documents with ID/IDREF links. The third extension could be on the property computed by the algorithm. Here, the property is the safety for selection and rejection. But the core of the algorithm consists in putting the interesting information from the context (the states of ancestors, typically) into the current state, so that the algorithm can use it progressively.

Chapter 6

Streamable Fragments of Forward XPath

Contents

6.1 Introduction

Forward XPath is not streamable, even if restricted to downward axes, as we have seen in Chapter 3 (Corollary 3). In this chapter, we distinguish fragments of Forward XPath that are m-streamability for all $m \in \mathbb{N}_0$. A streaming algorithm is obtained by compilation to dSTAs in PTIME. Here, we overcome the difficulty that Vardi and Wolper's automata construction for formulas of the modal logic LTL [VW94] and thus for XPath [LS08] may produce non-deterministic tree automata of exponential size. In contrast, our construction yields deterministic tree automata of polynomial size.

This chapter illustrates that dSTAs guide us towards relevant restrictions on Forward XPath. We conjecture that most of our restrictions are indeed neces-

sary for streamability and thus independent of our automata approach. While our results can be understood as a proof of concept, they do not yet constitute an exhaustive treatment with narrow upper and lower bounds.

Our translation will be by induction on the structure of path expressions. For simplicity, we consider the fragment of Forward XPath with downward axes *ch* and *ch** only. Our construction requires the following syntactic and semantic restrictions (based on the schema), which define the query language k-Downward XPath for $k \in \mathbb{N}$.

First of all, the usage of intersections needs to be limited, which arise when translating conjunctions in path expressions. Allowing an unbounded number of conjunctions would correspond to intersecting an unbounded number of automata, and thus require exponential time. As we need a PTIME translation, we limit the number of branches of k-Downward XPath expressions to k.

Second, non-determinism must be avoided when translating descendant axis *ch**, since otherwise, simultaneous treatments of all possible matches may overlap. Suppose for instance, that we want to construct a dSTA for a path expression *ch**::$*$[*F*] from a dSTA A_F for filter *F*. Then, for each descendant of the root, we have to run A_F. This can lead to an unbounded number of simultaneous runs of A_F to be handled by A, so that A cannot be of polynomial size. In order to avoid such overlaps, we require that all steps with descendant axis are guarded by a node label, i.e., they must have the form *ch**::a[*F*]. Furthermore, we impose the semantic restriction, that no tree satisfying the schema may contain nested a-nodes. This way, there exists at most one a-node per branch of every valid tree, so that we can check them by independent runs of A_F on all subtrees rooted at a-nodes. Automaton A starts by looking for an a-node, and once such a node is found, it runs the automaton A_F in order to check whether this a-node verifies F. When closing the a-node, the automaton checks whether the run of A_F was successful, and searches for another a-node on another branch if A_F failed.

Based on these restrictions, we obtain a translation of k-Downward XPath expressions to equivalent dSTAs in PTIME. Combined with the earliest query answering (EQA) algorithm for dSTAs of Chapter 5, this translation yields an EQA algorithm for k-Downward XPath and proves m-streamability for all $m \in \mathbb{N}_0$, but not ∞-streamable, since k-Downward XPath contains queries with unbounded concurrency.

Even though k-Downward XPath is small in that it supports only downward axes, it is still very expressive, as it allows for conjunctions, disjunctions, negations, and supports n-ary queries. The restrictions of k-Downward XPath are natural, in that they avoid overlapping tests of the same filter for different matches. We conjecture that our approach can be extended to further axis, but that removing some of these other restrictions would lead to non-streamability. In the last section of this chapter, we discuss some opportunities for extensions and improvements.

First, we present a further restriction on k-Downward XPath, that should imply ∞-streamability, and second, we discuss a generalization with horizontal axes.

Related work The idea of translating XPath expressions into automata for streaming XPath evaluation has been proposed for a long time. Altinel and Franklin [AF00] proposed a translation of non-branching downward path expressions to word automata on the language of branches. Green et al. [GMOS03, GGM+04] also use this kind of translation, while allowing branching expressions, and using a stack during the evaluation.

Gupta and Suciu [GS03b] propose the use of deterministic pushdown automata, and come up with an algorithm that is closer to ours. In terms of complexity, the algorithm by Gupta and Suciu requires exponential time in the size of the query, as determinization is needed. Only needed parts of the automaton are determinized, though, as the algorithm computes it lazily. Moreover, their fragment subsumes k-Downward XPath, as it mainly consists in CoreXPath 1.0, with downward axes and data joins.

Compact representations of automata were also investigated, in the context of XPath streaming evaluation. Transducer networks are such compact representations. They consist in a network of pushdown transducers, that are pushdown automata sending messages to other automata. Translations of several XPath fragments to transducer networks were investigated. Peng and Chawathe [PC05] focus on XPath with downward axes, while Olteanu [Olt07b] translates all of Forward XPath. Benedikt and Jeffrey [BJ07] study the filtering case for a fragment of XPath where matching can be decided at opening (resp. closing) time. Benedikt, Jeffrey and Ley-Wild [BJLW08] prove that this translation can be done in linear space and time for a fragment using backward guarded moves. More generally, all the aforementioned translations of XPath fragments to transducer networks are in PTIME and yield time-efficient algorithms. However, transducer networks are not adapted to static analysis, and all these algorithms store useless candidates in some cases. In [BJLW08], Benedikt, Jeffrey and Ley-Wild propose to replace transducer networks by binary decision diagrams (BDDs [Bry86]), as these can also be used as compact data structures for automata. Translations of transducer networks and BDDs to standard automata are in exponential time, so that we cannot use these representations to get a PTIME EQA algorithm using the algorithm for dSTAs in Chapter 5.

XPath is a navigational language, whose similarities with modal logics has been extensively studied [Lib06]. LTL, the Linear Temporal Logic, is a modal logic defining properties over words, using modality operators *Next*, *Previous*, *Until* and *Since*. A variant of LTL for tree structures, called $\mathrm{TL}_{\mathrm{tree}}$, has been proposed by Schlingloff [Sch92], and XPath expressions can be translated in linear

time to equivalent TL_{tree} formulas [Mar05a].

Vardi and Wolper [VW94] propose a translation of LTL formulas to automata in exponential time, for infinite words. This construction can be easily adapted for TL_{tree} over finite trees. Libkin and Sirangelo [LS08] propose such a translation from TL_{tree} formulas into query automata [NS02], i.e. tree automata using word automata to recognize the languages of labels of children. This translation also uses exponential time. Calvanese et al. [CDGLV09] proved recently that Regular XPath can also be translated in exponential time to non-deterministic tree automata (standard automata over *fcns* encodings of trees). This time, the authors do not use modal logics as intermediate query language, but alternating tree automata.

CoreXPath 1.0 has the expressiveness as the two-variables fragment of FO over trees [MdR05], and is thus strictly less expressive than MSO and tree automata. Using the standard techniques for translating MSO formulas to tree automata [Don70, TW68] leads to algorithms with non-elementary complexity [KMV07, Cla08].

6.2 m-Streamable Fragments of Forward XPath

We start this chapter by introducing m-streamable XPath fragments, for $m \in \mathbb{N}_0$. We define k-Downward XPath by imposing semantic and syntactic restrictions simultaneously. The expressions of k-Downward XPath are pairs of definitions of n-ary queries and schemas. Schemas are defined by dSTAs and queries by filters terms with n variables. Using filter terms with variables instead of Forward XPath expressions is not essential, but has the advantage of being more general while simplifying algorithms. In the remainder of the chapter, we assume that $|\Sigma| \geq 2$.

6.2.1 Filter Terms with Variables

Let $\mathcal{D} = \{ch^*, ch\}$ be the set of axis and $\mathcal{V}$ a set of variables. Filter terms are ranked trees with signature $\Delta = \{\mathit{and}, \mathit{not}, \mathit{true}, /, *\} \cup \mathcal{D} \cup \Sigma \cup \mathcal{V}$ as below, where $d \in \mathcal{D}$, $\ell \in \Sigma \cup \{*\}$ and $x \in \mathcal{V}$.

$$T \;::=\; \mathit{and}(T_1, T_2) \mid \mathit{not}(T) \mid \mathit{true} \mid /(T) \mid d(T) \mid \ell(T) \mid x$$

The only additional restriction we assume, is that the operator $/$ can appear in root position only. Terms of the form $/(T)$ correspond to root filters and all others to ordinary filters. Given a tree t and a variable assignment $\mu : \mathcal{V} \to \mathit{nod}(t)$, we define a set valued semantics $[\![T]\!]_{t,\mu} \subseteq \mathit{nod}(t)$ for all filter terms in Figure 6.1.

$$\begin{array}{ll}
[\![/T]\!]_{t,\mu} = \{\epsilon\} \cap [\![T]\!]_{t,\mu} & [\![d(T)]\!]_{t,\mu} = \{\pi \mid \exists \pi' \in [\![T]\!]_{t,\mu}.\ (\pi,\pi') \in d^t\} \\
[\![x]\!]_{t,\mu} = \{\mu(x)\} & [\![\ell(T)]\!]_{t,\mu} = \{\pi \mid \ell \in \{*, \mathit{lab}^t(\pi)\}\} \cap [\![T]\!]_{t,\mu} \\
[\![\mathit{true}]\!]_{t,\mu} = \mathit{nod}(t) & [\![\mathit{and}(T_1,T_2)]\!]_{t,\mu} = [\![T_1]\!]_{t,\mu} \cap [\![T_2]\!]_{t,\mu} \\
[\![\mathit{not}(T)]\!]_{t,\mu} = \mathit{nod}(t) - [\![T]\!]_{t,\mu} &
\end{array}$$

Figure 6.1: Semantics of filter terms.

$$\begin{array}{ll}
\mathfrak{T}([\mathit{self}::\ell]) = \ell(\mathit{true}) & \mathfrak{T}([d::\ell]) = d(\ell(\mathit{true})) \\
\mathfrak{T}([\mathit{self}::\ell\, F]) = \ell(\mathfrak{T}(F)) & \mathfrak{T}([d::\ell\, F]) = d(\ell(\mathfrak{T}(F))) \\
\mathfrak{T}([\mathit{self}::\ell/P]) = \ell(\mathfrak{T}([P])) & \mathfrak{T}([d::\ell/P]) = d(\ell(\mathfrak{T}([P]))) \\
\mathfrak{T}([not(F)]) = not(\mathfrak{T}(F)) & \mathfrak{T}([F_1\ and\ F_2]) = and(\mathfrak{T}(F_1), \mathfrak{T}(F_2)) \\
\mathfrak{T}([x]) = x & \mathfrak{T}(/P) = /(\mathfrak{T}([P]))
\end{array}$$

Figure 6.2: Filters and rooted paths as filter terms, where $d \neq \mathit{self}$. We assume the selection position of rooted paths was marked at beforehand by a variable $[x]$.

In Figure 6.2, we map XPath filters and rooted paths using axes $\{\mathit{self}, \mathit{ch}, \mathit{ch}^*\}$ to filter terms. The translation of filters $\mathfrak{T}(F)$ is straightforward. Similarly, we translate rooted paths R to filter terms $\mathfrak{T}(R(x))$ with a single free variable x. We annotate this variable before translation to R by using the extra filter $[x]$. The translation preserves the semantics: For filters, we have

$$[\![F]\!]^t_{\text{filter}} = [\![\mathfrak{T}(F)]\!]_{t,\mu}$$

for all variable assignments μ. For root filters R, where x annotates the selection position, we have

$$[\![R(x)]\!]^t_{\text{filter}} = \{\mu(x) \mid [\![\mathfrak{T}(R(x))]\!]_{t,\mu} \neq \emptyset\}$$

6.2.2 k-Downward XPath

Let the *width* of a term T be the number of its leaves. This corresponds to the maximum number of conjunctions to be tested simultaneously. We have to bound this number for our automata constructions (condition 1 below).

Descendant axis are a source of trouble since they are highly nondeterministic. The query defined by $/(\mathit{ch}^*(*(\mathit{and}(x, \mathit{ch}^*(a)))))$ for instance has unbounded concurrency, since the selection of b-nodes in trees $b(b(b(\ldots(a)\ldots)))$ can be decided only when encountering the a-leaf. This problem is solved by three restrictions:

All descendant steps must be guarded by a label of Σ, i.e., they must all be of the form $\mathit{ch}^*(a(T))$ (condition 3). We impose a semantic restriction on all trees t acceptable by the schema, stating that no further a-node may be encountered below an a-node in t (condition 4). All filters must start at the root, in order to avoid any implicit descending step (condition 2). Finally, we only consider shallow trees (condition 5).

Let $k \in \mathbb{N}$. We define *k-Downward XPath* as the query class containing all pairs $(T(x_1, \ldots, x_n), B)$ of terms T with a sequence of variables $x_1, \ldots, x_n$ and dSTAs B with signature Σ, that satisfy the following conditions:

1. the width of T is bounded by k, i.e., T has at most k leaves.
2. T starts at the root, i.e., T matches some term $/(T')$.
3. if $\mathit{ch}^*(T')$ is a subterm of T then T' matches some term $a(T'')$.
4. if $\mathit{ch}^*(a(T''))$ is a subterm of T then:

$$\forall t \in L(B).\, \forall \pi, \pi' \in \mathit{lab}_a(t).\, \pi \neq \pi' \Rightarrow \neg(\mathit{ch}^*)^t(\pi, \pi')$$

5. the depth of the valid trees $t \in L(B)$ is bounded by some constant.

6.2.3 Deciding Membership to k-Downward XPath

A procedure for testing in PTIME whether a pair $(T(x_1, \ldots, x_n), B)$ is in k-Downward XPath can be obtained. We first characterize STAs recognizing trees of bounded depth, in order to decide condition 5.

Lemma 13. *For fixed n, it can be decided in* PTIME *whether an STA B accepts trees of bounded depth, i.e., whether $\exists d \in \mathbb{N}.\, t \in L(B) \Rightarrow \mathit{depth}(t) \leq d$.*

Proof. To decide whether trees in $L(B)$ are of bounded depth, we look for vertical loops. Let *deep* be the relation on $(\mathit{stat}_{\mathrm{e}}^B)^4$ defined by:

$$\mathit{deep}(q_1, q_1', q_2, q_2') \Leftrightarrow \left\{ \begin{array}{l} \exists t \in \mathcal{T}_\Sigma.\, \exists r \in \mathit{runs}^B(t).\, \exists (\pi, \pi') \in \mathit{ch}^+(t). \\ \quad q_1 = r(\mathit{pr}((\mathit{op}, \pi))) \ \wedge\ q_1' = r((\mathit{cl}, \pi)) \wedge \\ \quad q_2 = r(\mathit{pr}((\mathit{op}, \pi'))) \ \wedge\ q_2' = r((\mathit{cl}, \pi')) \end{array} \right.$$

The relation *deep* can be computed by the Datalog program given by inference rules in Figure 6.3. We use a smooth notation: rules in hypothesis of inference schemas have to be rules of B, and $\mathit{acc}_{\mathcal{H}_\Sigma}$ is the accessibility relation of B through hedges on alphabet Σ, as defined in Section 5.4.2. The first inference schema handles the case where π' is a child of π. The second one is the recursive case

$$\frac{q_1 \xrightarrow{\mathit{op}\ a:\gamma_1} q_3 \xrightarrow{\mathit{acc}_{\mathcal{H}_\Sigma}} q_2 \xrightarrow{\mathit{op}\ b:\gamma_2} q_4 \xrightarrow{\mathit{acc}_{\mathcal{H}_\Sigma}} q_5 \xrightarrow{\mathit{cl}\ b:\gamma_2} q_2' \xrightarrow{\mathit{acc}_{\mathcal{H}_\Sigma}} q_6 \xrightarrow{\mathit{cl}\ a:\gamma_1} q_1'}{\mathit{deep}(q_1, q_1', q_2, q_2').}$$

$$\frac{q_1 \xrightarrow{\mathit{op}\ a:\gamma_1} q_3 \xrightarrow{\mathit{acc}_{\mathcal{H}_\Sigma}} q_4 \qquad q_4' \xrightarrow{\mathit{acc}_{\mathcal{H}_\Sigma}} q_6 \xrightarrow{\mathit{cl}\ a:\gamma_1} q_1' \qquad q_2, q_2' \in \mathit{stat}_{\mathrm{e}}}{\mathit{deep}(q_1, q_1', q_2, q_2') \text{ :- } \mathit{deep}(q_4, q_4', q_2, q_2').}$$

Figure 6.3: Inference rules for the definition of *deep*.

for deeper depths. We call a state *productive* if it can be reached by an initial state, and a final state can be reached from it. $L(B)$ is not of bounded depth iff we can loop between (q_1, q_1') and (q_1, q_1') itself, i.e. iff there exists productive states $q_1, q_1' \in \mathit{stat}_{\mathrm{e}}^B$ such that $\mathit{deep}(q_1, q_1', q_1, q_1')$. This can be checked in PTIME. □

Proposition 22. *Given a term $T(x_1, \ldots, x_n)$ and a dSTA B over Σ, it is decidable in* PTIME *depending on $|T|$, $|B|$, $|\Sigma|$, k and n whether $(T(x_1, \ldots, x_n), B)$ is in k-Downward XPath.*

Proof. Conditions 1 to 3 are syntactic, and can be checked in PTIME in $|T|$. Condition 5 can be checked in PTIME in $|B|$ by Lemma 13. For condition 4, let D_a be a dSTA accepting trees having two a-nodes in a branch, i.e.: $L(D_a) = \{t \in \mathcal{T}_\Sigma \mid \exists \pi, \pi' \in \mathit{lab}_a^t.\ \mathit{ch}^+(\pi, \pi')\}$. Then, for every label $a \in \Sigma$ such that $\mathit{ch}^*(T')$ is a subterm of T for some T', we have to check that $L(B) \cap L(D_a) = \emptyset$. This can be done in time $O(|T| \cdot |B| \cdot |\Sigma|)$. □

6.2.4 Translating k-Downward XPath to dSTAs

For fixed $k \in \mathbb{N}$, we propose a new PTIME translation of expressions $(T(x_1, \ldots, x_n), B)$ of k-Downward XPath into dSTAs (A, B) such that $(T(x_1, \ldots, x_n), B)$ and (A, B) both recognize the same query $Q_{A,B}$. The dSTA B defining the schema does not need translation, and we only compile terms $T(x_1, \ldots, x_n)$ into dSTAs A. The translation is correct and in PTIME if B is such that $(T(x_1, \ldots, x_n), B)$ is in k-Downward XPath.

For clarity, we first provide a translation of expressions in k-Downward XPath to dSTAs such that the target dSTAs accept non-canonical trees: variables in $\mathcal{V}_n$ may not appear, or appear several times in those trees. For this purpose, we extend canonical annotations. For a tree $t \in \mathcal{T}_\Sigma$ and a function ν: $\mathit{nod}(t) \to 2^{\mathcal{V}_n}$, let $t \tilde{*} \nu$ be the tree with $\mathit{dom}(t \tilde{*} \nu) = \mathit{dom}(t)$ and for all nodes $\pi \in \mathit{nod}(t)$, $\mathit{lab}^{t \tilde{*} \nu}(\pi) = (\mathit{lab}^t(\pi), \nu(\pi))$. The semantics of filter terms is extended in the natural way, by changing the semantics of variables $x \in \mathcal{V}_n$: $[\![x]\!]_{t,\nu} = \{\pi \in \mathit{nod}(t) \mid x \in \nu(\pi)\}$.

Moreover, dSTAs resulting from the translation are such that there exists a run on every tree over $\Sigma \times 2^{\mathcal{V}_n}$. STAs having these property are called *pseudo-complete* in the sequel.

Lemma 14. *There exists $c > 0$ such that for every expression $(T_1(x_1, \ldots, x_n), B)$ of k-Downward XPath and subterm T of T_1, a pseudo-complete dSTA A over signature $\Sigma \times 2^{\mathcal{V}_n}$ with at most $(3 \cdot |T|)^{\mathit{width}(T)}$ event and node states can be computed in time at most $c \cdot (|\mathit{rul}^A| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|)$ such that for every tree $t \in L(B)$ and ν: $\mathit{nod}(t) \to 2^{\mathcal{V}_n}$:*

$$t\tilde{*}\nu \in L(A) \qquad \textit{iff} \qquad \epsilon \in [\![T]\!]_{t,\nu}$$

Proof. The proof is by induction on the structure of filter terms. For $\alpha \in \{\mathit{op}, \mathit{cl}\}$ and $a \in \Sigma$, we write $\mathit{rul}^A_{\alpha,a} = \{q_1 \xrightarrow{\alpha\ a:\gamma} q_2 \in \mathit{rul}^A\}$. In the following, we assume that dSTAs are stored by a data structure for which we can find constants c_i ($1 \leq i \leq 7$) such that:

(i). for every pair of pseudo-complete dSTAs (A_1, A_2), a pseudo-complete dSTA for $A_1 \cap A_2$ with $|\mathit{stat}_{\mathrm{e}}^{A_1}| \cdot |\mathit{stat}_{\mathrm{e}}^{A_2}|$ event states, $|\mathit{stat}_{\mathrm{n}}^{A_1}| \cdot |\mathit{stat}_{\mathrm{n}}^{A_2}|$ node states and such that $|\mathit{rul}^A| = \sum_{\alpha \in \{\mathit{op},\mathit{cl}\}, a \in \Sigma} |\mathit{rul}^{A_1}_{\alpha,a}| \cdot |\mathit{rul}^{A_2}_{\alpha,a}|$ can be computed in time $c_1 \cdot |\mathit{rul}^A|$.

(ii). for every dSTA A, the dSTA A' obtained by swapping the final states of A (i.e., $\mathit{fin}^{A'} = \mathit{stat}_{\mathrm{e}}^A - \mathit{fin}^A$) can be obtained in constant time c_2.[1]

(iii). for every dSTA A, the set of rules $I_A = \{q_1 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2 \in \mathit{rul}^A \mid q_1 \in \mathit{init}^A\}$ can be computed in time $c_3 \cdot |I_A|$.

(iv). for every dSTA A and every $(a, v, \gamma) \in \Sigma \times 2^{\mathcal{V}_n} \times \mathit{stat}_{\mathrm{n}}^A$, the set of closing rules $C_A = \{q_1 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^A\}$ can be computed in time $c_4 \cdot |C_A|$.

(v). for every dSTA A and symbol $a \in \Sigma$, we can build in constant time c_5 the dSTA A' obtained from A by removing all rules using a, i.e. $\mathit{rul}^{A'} = \mathit{rul}^A - \{q_1 \xrightarrow{\alpha\ (a,v):\gamma} q_2 \in \mathit{rul}^A\}$.[2]

(vi). for every dSTA A and symbol $a \in \Sigma$, the set of rules $R_A = \{q_1 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^A\}$ can be computed in time $c_6 \cdot |R_A|$.

[1] This can be achieved, for instance, with one flag for the automaton, indicating whether the set of final states has to be interpreted as its complement.

[2] We assume in the sequel that $c_5 \geq 2$.

(vii). for every dSTA A and states q_0, q_1, q_2, γ, the following sequence of operations can be performed in constant time c_7: add q_0 to $\mathit{stat}_{\mathrm{e}}^A$, add γ to $\mathit{stat}_{\mathrm{n}}^A$, set init^A to $\{q_1\}$, and set fin^A to $\{q_2\}$.

Let us now prove the invariant by induction on the structure of T, with $c = \frac{c_1}{2} + c_2 + c_3 + c_4 + c_5 + c_6 + 3 \cdot c_7$. The time needed for building a rule (given all its parameters) and adding it to the set of rules of a dSTA is supposed to be 1 in the following.

Case $T = \mathit{and}(T_1, T_2)$**.** Let A_1 be the pseudo-complete dSTA for T_1 and A_2 the pseudo-complete dSTA for A_2. Let A be the product of the two, such that pairs of final states are accepting, and pairs of initial states are initial. A recognizes the correct tree language, as for all trees $t \in L(B)$ and $\nu\colon \mathit{nod}(t) \to 2^{\mathcal{V}_n}$:

$$\begin{array}{rcll} t\tilde{*}\nu \in L(A) & \Leftrightarrow & t\tilde{*}\nu \in L(A_1) \ \wedge \ t\tilde{*}\nu \in L(A_2) & \\ & \Leftrightarrow & \epsilon \in [\![T_1]\!]_{t,\nu} \ \wedge \ \epsilon \in [\![T_2]\!]_{t,\nu} & \text{by induction hypothesis} \\ & \Leftrightarrow & \epsilon \in [\![T]\!]_{t,\nu} & \end{array}$$

A is deterministic and pseudo-complete since A_1 and A_2 are deterministic and pseudo-complete. The number of event states of A is:

$$\begin{array}{rcll} |\mathit{stat}_{\mathrm{e}}^A| & = & |\mathit{stat}_{\mathrm{e}}^{A_1}| \cdot |\mathit{stat}_{\mathrm{e}}^{A_2}| & \\ & \leq & (3 \cdot |T_1|)^{\mathit{width}(T_1)} \cdot (3 \cdot |T_2|)^{\mathit{width}(T_2)} & \text{by induction hypothesis} \\ & \leq & (3 \cdot |T|)^{\mathit{width}(T_1)} \cdot (3 \cdot |T|)^{\mathit{width}(T_2)} & \\ & \leq & (3 \cdot |T|)^{\mathit{width}(T_1)+\mathit{width}(T_2)} & \\ & \leq & (3 \cdot |T|)^{\mathit{width}(T)} & \end{array}$$

and similarly for node states. Building A consists in building A_1 and A_2 (which can be done in time $c \cdot (|\mathit{rul}^{A_1}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T_1)} + |T_1|) + c \cdot (|\mathit{rul}^{A_2}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T_2)} + |T_2|)$ by induction hypothesis) and then A from these two dSTAs, which can be done in time $c_1 \cdot |\mathit{rul}^A|$ by condition (i). Hence the total time for building A is:

$$\begin{array}{l} \quad c{\cdot}(|\mathit{rul}^{A_1}|{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T_1)}{+}|T_1|) \\ + \, c{\cdot}(|\mathit{rul}^{A_2}|{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T_2)}{+}|T_2|) + c_1{\cdot}|\mathit{rul}^A| \\ = \Theta + c{\cdot}(|T_1| + |T_2|) + c_1{\cdot}|\mathit{rul}^A| \end{array}$$

with

$$\begin{array}{rcll} \Theta = & & c{\cdot}(|\mathit{rul}^{A_1}|{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T_1)}) & \\ & + & c{\cdot}(|\mathit{rul}^{A_2}|{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T_2)}) & \\ & \leq & c{\cdot}(|\mathit{rul}^{A_1}| + |\mathit{rul}^{A_2}|){\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T)-1} & \text{as } \mathit{width}(T){-}1 \geq \mathit{width}(T_i) \\ & \leq & c{\cdot}|\mathit{rul}^A|{\cdot}\frac{|\mathit{rul}^{A_1}|+|\mathit{rul}^{A_2}|}{|\mathit{rul}^A|}{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T)-1} & \\ & \leq & c{\cdot}|\mathit{rul}^A|{\cdot}4{\cdot}|\Sigma|{\cdot}(5{\cdot}|\Sigma|)^{\mathit{width}(T)-1} & \text{cf below} \end{array}$$

For the last inequality, we know from condition (i) that $|\mathit{rul}^A| = \sum_{\alpha\in\{op,cl\},a\in\Sigma} |\mathit{rul}^{A_1}_{\alpha,a}| \cdot |\mathit{rul}^{A_2}_{\alpha,a}|$. By grouping by actions and letters, we get:

$$\begin{aligned}\frac{|\mathit{rul}^{A_1}|+|\mathit{rul}^{A_2}|}{|\mathit{rul}^A|} &= \frac{|\mathit{rul}^{A_1}|+|\mathit{rul}^{A_2}|}{\sum_{\alpha\in\{op,cl\},a\in\Sigma} |\mathit{rul}^{A_1}_{\alpha,a}|\cdot|\mathit{rul}^{A_2}_{\alpha,a}|} = \frac{\sum_{\alpha\in\{op,cl\},a\in\Sigma} |\mathit{rul}^{A_1}_{\alpha,a}|+|\mathit{rul}^{A_2}_{\alpha,a}|}{\sum_{\alpha\in\{op,cl\},a\in\Sigma} |\mathit{rul}^{A_1}_{\alpha,a}|\cdot|\mathit{rul}^{A_2}_{\alpha,a}|}\\ &\leq \sum_{\alpha\in\{op,cl\},a\in\Sigma} \frac{|\mathit{rul}^{A_1}_{\alpha,a}|+|\mathit{rul}^{A_2}_{\alpha,a}|}{|\mathit{rul}^{A_1}_{\alpha,a}|\cdot|\mathit{rul}^{A_2}_{\alpha,a}|} \leq 4\cdot|\Sigma|\end{aligned}$$

Note that $|\mathit{rul}^{A_1}_{\alpha,a}| > 0$ and $|\mathit{rul}^{A_2}_{\alpha,a}| > 0$ as A is pseudo-complete. Finally, the total time for computing A is:

$$\begin{aligned}&\Theta + c\cdot(|T_1|+|T_2|) + c_1\cdot|\mathit{rul}^A|\\ \leq\ &\Theta + c\cdot|T| + c_1\cdot|\mathit{rul}^A| && \text{as } |T|{=}|T_1|{+}|T_2|{+}1\\ \leq\ &c\cdot|\mathit{rul}^A|\cdot 4\cdot|\Sigma|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)-1} + c\cdot|T| + c_1\cdot|\mathit{rul}^A|\\ \leq\ &c\cdot(|\mathit{rul}^A|\cdot((5\cdot|\Sigma|)^{\mathit{width}(T)-1}\cdot(4\cdot|\Sigma|) + \tfrac{c_1}{c}) + |T|)\\ \leq\ &c\cdot(|\mathit{rul}^A|\cdot((5\cdot|\Sigma|)^{\mathit{width}(T)-1}\cdot(4\cdot|\Sigma| + \tfrac{c_1}{c})) + |T|)\\ \leq\ &c\cdot(|\mathit{rul}^A|\cdot((5\cdot|\Sigma|)^{\mathit{width}(T)-1}\cdot 5\cdot|\Sigma|) + |T|) && \text{as } \tfrac{c_1}{c} \leq 2 \leq |\Sigma|\\ \leq\ &c\cdot(|\mathit{rul}^A|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T|)\end{aligned}$$

Case $T = \mathit{not}(T')$**.** Let A' be the pseudo-complete dSTA built for T'. Let A be the STA obtained from A' by swapping the final states, i.e.:

$$\begin{array}{lll}\mathit{stat}^A_{\mathrm{e}} = \mathit{stat}^{A'}_{\mathrm{e}} & \mathit{init}^A = \mathit{init}^{A'} & \mathit{rul}^A = \mathit{rul}^{A'}\\ \mathit{stat}^A_{\mathrm{n}} = \mathit{stat}^{A'}_{\mathrm{n}} & \mathit{fin}^A = \mathit{stat}^{A'}_{\mathrm{e}} - \mathit{fin}^{A'} & \end{array}$$

A' is deterministic and pseudo-complete, so we get:

$$t\tilde{*}\nu \in L(A) \quad\Leftrightarrow\quad t\tilde{*}\nu \notin L(A') \overset{\text{ind. hyp.}}{\Leftrightarrow} \epsilon \notin [\![T']\!]_{t,\nu} \quad\Leftrightarrow\quad \epsilon \in [\![T]\!]_{t,\nu}$$

The number of states of A is:

$$\begin{aligned}|\mathit{stat}^A_{\mathrm{e}}| = |\mathit{stat}^{A'}_{\mathrm{e}}| &\leq (3\cdot|T'|)^{\mathit{width}(T')} && \text{by induction hypothesis}\\ &\leq (3\cdot|T|)^{\mathit{width}(T')}\\ &\leq (3\cdot|T|)^{\mathit{width}(T)}\end{aligned}$$

and similarly for $|\mathit{stat}^A_{\mathrm{e}}|$. By induction hypothesis, building A' can be done in time $c\cdot(|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T'|)$. By condition (ii), the time for building A from A' is bounded by c_2, so the total time for building A is at most:

$$\begin{aligned}&c\cdot(|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T'|) + c_2\\ =\ &c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T'|) + c_2\\ =\ &c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T'|) + c_2 && \text{as } \mathit{width}(T) = \mathit{width}(T')\\ \leq\ &c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T'|) + c && \text{as } c \geq c_2\\ \leq\ &c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T'| + 1)\\ \leq\ &c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T|) && \text{as } |T| = |T'|{+}1\end{aligned}$$

Case $T = \textbf{\textit{true}}$**.** As $[\![\mathit{true}]\!]_{t,\nu} = \mathit{nod}(t)$, A is universal, and we can build A as follows:

$$\mathit{stat}_{\mathrm{e}}^A = \mathit{stat}_{\mathrm{n}}^A = \mathit{init}^A = \mathit{fin}^A = \{1\} \qquad \frac{a \in \Sigma \quad v \subseteq \mathcal{V}_n \quad \alpha \in \{\mathit{op}, \mathit{cl}\}}{1 \xrightarrow{\alpha\ (a,v):1} 1 \in \mathit{rul}^A}$$

Obviously, $|\mathit{stat}_{\mathrm{e}}^A| = |\mathit{stat}_{\mathrm{n}}^A| \leq (3 \cdot |T|)^{\mathit{width}(T)}$. Building A requires time at most c_7 for setting the states (by condition (vii)), plus time $|\mathit{rul}^A|$ for the rules, so a total time of:

$$\begin{array}{rcll} |\mathit{rul}^A| + c_7 & \leq & c \cdot |\mathit{rul}^A| + c_7 & \\ & \leq & c \cdot (|\mathit{rul}^A| + 1) & \text{as } c \geq c_7 \\ & \leq & c \cdot (|\mathit{rul}^A| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + 1) & \\ & \leq & c \cdot (|\mathit{rul}^A| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \end{array}$$

Case $T = /(T')$**.** By definition, $\epsilon \in [\![T]\!]_{t,\nu} \Leftrightarrow \epsilon \in [\![T']\!]_{t,\nu}$ so we can keep the automaton for T'.

Case $T = \textbf{\textit{ch}}(T')$**.** Let A' be the automaton built for T'. The automaton A for T has to launch A' when opening each child of the root. Here we need three additional event states $\mathit{stat}_{\mathrm{e}}^A = \mathit{stat}_{\mathrm{e}}^{A'} \uplus \{\mathit{start}, 0, 1\}$: *start* is only used as initial state, to detect (op, ϵ), while 0 and 1 are used between the children of the root, to propagate the detection of matchings: $\mathit{init}^A = \{\mathit{start}\}$ and $\mathit{fin}^A = \{1\}$. We also need two new node states, in order to pass information about matchings through children of the root: $\mathit{stat}_{\mathrm{n}}^A = \mathit{stat}_{\mathrm{n}}^{A'} \uplus \{0, 1\}$. We detect the last event (cl, ϵ) by the fact that we close from an event state in $\{0, 1\}$, if the root has children. Otherwise, we close in state 0, so the run will not be accepting. We define the rules of A by the following inference schemas:

$$\frac{a \in \Sigma \qquad v \subseteq \mathcal{V}_n}{\mathit{start} \xrightarrow{\alpha\ (a,v):0} 0}$$ opening the root: move to 0

$$\frac{q_1 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'} \qquad q_1 \in \mathit{init}^{A'} \quad \flat \in \{0,1\}}{\flat \xrightarrow{\mathit{op}\ (a,v):\ \flat} q_2}$$ opening a child: start testing T'

$$\frac{q_1 \xrightarrow{\alpha\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'}}{q_1 \xrightarrow{\alpha\ (a,v):\gamma} q_2}$$ run test of T'

$$\frac{q_1 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'} \qquad q_2 \notin \mathit{fin}^{A'} \qquad \flat \in \{0,1\}}{q_1' \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2' \in \mathit{rul}^{A'} \qquad q_1' \in \mathit{init}^{A'}} \quad q_1 \xrightarrow{\mathit{cl}\ (a,v):\ \flat} \flat$$

failure of T': no new match

$$\frac{q_1 \xrightarrow{\mathit{cl}\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'} \qquad q_2 \in \mathit{fin}^{A'} \qquad \flat \in \{0,1\}}{q_1' \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2' \in \mathit{rul}^{A'} \qquad q_1' \in \mathit{init}^{A'}} \quad q_1 \xrightarrow{\mathit{cl}\ (a,v):\ \flat} 1$$

success of T': move to 1

$$\frac{a \in \Sigma \qquad v \subseteq \mathcal{V}_n \qquad \flat \in \{0,1\}}{\flat \xrightarrow{\mathit{cl}\ (a,v):0} \flat}$$

closing the root

A is deterministic. The fact that all axes in $\mathcal{D}$ are downwards permits to decide, when closing a child, whether this child matches T'. By a left-to-right induction on the children of the root of $t * \nu$, we can prove that the run r of A on $t * \nu$ assigns 1 to (cl, i) if there is an accepting run of A' on a child j (with $1 \leq j \leq j$) of ϵ, and 0 otherwise. As this Boolean is kept when closing the root, and is set to 0 if there is no child, we have:

$$\begin{array}{rcl} a(t_1, \ldots, t_k) * \nu \in L(A) & \Leftrightarrow & \exists 1 \leq i \leq k.\ t_i * \nu_i \in L(A') \\ & \overset{\text{ind. hyp.}}{\Leftrightarrow} & \exists 1 \leq i \leq k.\ \epsilon \in [\![T']\!]_{t_i,\nu_i} \\ & \Leftrightarrow & \epsilon \in [\![T]\!]_{t,\nu} \end{array}$$

where ν_i is the restriction of ν to nodes of t_i. Moreover, we just introduced three event states:

$$\begin{array}{rcll} |\mathit{stat}_{\mathrm{e}}^{A}| = |\mathit{stat}_{\mathrm{e}}^{A'}| + 3 & \leq & (3 \cdot |T'|)^{\mathit{width}(T')} + 3 & \text{by induction hypothesis} \\ & \leq & (3 \cdot |T'|)^{\mathit{width}(T)} + 3 & \\ & \leq & (3 \cdot (|T'| + 1))^{\mathit{width}(T)} & \\ & \leq & (3 \cdot |T|)^{\mathit{width}(T)} & \end{array}$$

and we only introduced two node states, which is even lower. In terms of time cost, we have to prove that every new rule is built in constant time. This is straightforward for the n_1 rules operating at the root. By condition (iii), the n_2 rules for opening a child are built in time $c_3 \cdot n_2$. For the j-th rule among these n_2 rules, we can compute p_j rules with corresponding labels and node states in time $c_4 \cdot p_j$, according to condition (iv). We include in c_4 the cost for testing whether the target state is final. Moreover, the time for adding the new states to $\mathit{stat}_{\mathrm{e}}^{A}$ and $\mathit{stat}_{\mathrm{n}}^{A}$, and setting initial and final states is bounded by $3 \cdot c_7$, according to condition (vii). Let Θ be the time for

computing A' and for setting initial and final states:

$$\begin{array}{rcll}
\Theta & = & c \cdot (|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T'|) + 3\cdot c_7 & \text{by induction hyp.} \\
& \leq & c \cdot (|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T'| + 1) & \text{as } 3\cdot c_7 \leq c \\
& \leq & c \cdot (|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T')} + |T|) & \text{as } |T| = |T'| + 1 \\
& \leq & c \cdot (|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T|) & \text{as } \mathit{width}(T') = \mathit{width}(T)
\end{array}$$

The total time for building A is:

$$\begin{array}{ll}
\Theta + n_1 + c_3 \cdot n_2 + \sum_{1\leq j\leq n_2} c_4 \cdot p_j & \\
= \Theta + n_1 + c_3 \cdot n_2 + c_4 \cdot \sum_{1\leq j\leq n_2} p_j & \\
\leq \Theta + (c_3 + c_4) \cdot (n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j) & \\
\leq \Theta + c \cdot (n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j) & \text{as } c_3 + c_4 \leq c \\
\leq c\cdot(|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T| + n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j) & \text{cf above} \\
\leq c\cdot(|\mathit{rul}^{A'}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j + |T|) & \\
\leq c\cdot((|\mathit{rul}^{A'}| + n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j)\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T|) & \\
\leq c\cdot(|\mathit{rul}^{A}|\cdot(5\cdot|\Sigma|)^{\mathit{width}(T)} + |T|) & \text{cf below}
\end{array}$$

The last inequality holds because $|\mathit{rul}^{A}| = |\mathit{rul}^{A'}| + n_1 + n_2 + \sum_{1\leq j\leq n_2} p_j$.

Case $T = \mathit{ch}^*(T'')$**.** By condition 3, $T'' = a(T')$ for some $a \in \Sigma$ and filter term T'. Let A' be the pseudo-complete dSTA constructed for T'. We define the pseudo-complete dSTA A for T as follows:

$$\begin{array}{ll}
\mathit{stat}^A_{\mathrm{e}} = \mathit{stat}^{A'}_{\mathrm{e}} \uplus \{0,1\} & \mathit{init}^A = \{0\} \\
\mathit{stat}^A_{\mathrm{n}} = \mathit{stat}^{A'}_{\mathrm{n}} \uplus \{0\} & \mathit{fin}^A = \{1\}
\end{array}$$

In event state 0, automaton A searches for an a-node matching T, while in event state 1 it has found such a node. Node state 0 is used everywhere except below a-nodes. At every time point there is at most one a-node to be considered, by condition 4.

$$\frac{b \in \Sigma - \{a\} \qquad v \subseteq \mathcal{V}_n \qquad \alpha \in \{\mathit{op}, \mathit{cl}\}}{0 \xrightarrow{\alpha\ (b,v):0} 0} \quad \text{wait for } a\text{-node}$$

$$\frac{q_1 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'} \qquad q_1 \in \mathit{init}^{A'}}{0 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2} \quad \text{find } a\text{-node: start testing } T'$$

$$\frac{b \in \Sigma - \{a\} \qquad q_1 \xrightarrow{\alpha\ (b,v):\gamma} q_2 \in \mathit{rul}^{A'}}{q_1 \xrightarrow{\alpha\ (b,v):\gamma} q_2} \quad \text{run test of } T'$$

$$\frac{q_1 \xrightarrow{cl\ (a,v):\gamma} q_2 \in rul^{A'} \qquad q_2 \notin fin^{A'}}{q_1 \xrightarrow{cl\ (a,v):\gamma} 0}$$ failure of T': restart

$$\frac{q_1 \xrightarrow{cl\ (a,v):\gamma} q_2 \in rul^{A'} \qquad q_2 \in fin^{A'}}{q_1 \xrightarrow{cl\ (a,v):\gamma} 1}$$ success of T'

$$\frac{b \in \Sigma \qquad v \subseteq \mathcal{V}_n \qquad \alpha \in \{op, cl\}}{1 \xrightarrow{\alpha\ (b,v):0} 1}$$ filter T successful

Now one can show how to construct a run of A for all trees verifying condition 4 such that $\epsilon \in [\![T]\!]_{t,\nu}$, and vice versa, if there exists a successful run of A on some tree $t * \nu$ verifying condition 4 then $\epsilon \in [\![T]\!]_{t,\nu}$.

One reason for which this works is that A' is pseudo-complete, so that the run for T' can always be continued. No match of a can be missed, since no node above a is labeled by a (condition 4). The only reason to move into a state different from 0 before opening the a-node is another a-node on the left. Either the run of T' there succeeds, and the automaton goes into the universal state 1, or else, it finishes but fails, and returns back into state 0, so that new a-nodes can be tested. Automaton A is deterministic, by determinism of A' and the inference schemas defining its rules. Moreover, A is pseudo-complete by construction. We obtain the following number of states:

$$\begin{aligned} |stat_{\mathrm{e}}^{A}| = |stat_{\mathrm{e}}^{A'}| + 2 &\leq (3 \cdot |T'|)^{width(T')} + 2 = (3 \cdot |T'|)^{width(T)} + 2 \\ &\leq (3 \cdot |T|)^{width(T)} \\ |stat_{\mathrm{n}}^{A}| = |stat_{\mathrm{n}}^{A'}| + 1 &\leq (3 \cdot |T'|)^{width(T')} + 1 = (3 \cdot |T'|)^{width(T)} + 1 \\ &\leq (3 \cdot |T|)^{width(T)} \end{aligned}$$

The time cost for building A can be decomposed as follows. Each of the n_1 rules waiting for an a-node, or propagating that filter T is successful, is generated in constant time. From condition (iii), the n_2 rules used when an a-node is found can be built in time $c_3 \cdot n_2$. The rules for testing T' are constructed in time c_5, according to condition (v). The n_3 rules used after a failure or success of T' are generated in time $c_6 \cdot n_3$, by condition (vi). Finally, the time needed for setting $stat_{\mathrm{e}}^{A}$, $stat_{\mathrm{n}}^{A}$, $init^{A}$ and fin^{A} is bounded by $2 \cdot c_7$, by condition (vii). Let Θ be the time for building A' plus the time

needed for setting $\mathit{stat}_{\mathrm{e}}^{A}$, $\mathit{stat}_{\mathrm{n}}^{A}$, init^{A} and fin^{A}:

$$\begin{array}{rcll}
\Theta & = & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T'|) + 2 \cdot c_7 & \text{by induction hypothesis} \\
& \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T'| + 1) & \text{as } 2 \cdot c_7 \leq c \\
& \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T|) & \text{as } |T| = |T'| + 2 \\
& \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \text{as } \mathit{width}(T) = \mathit{width}(T')
\end{array}$$

The overall time cost for computing A is:

$$\begin{array}{ll}
\Theta + n_1 + c_3 \cdot n_2 + c_5 + c_6 \cdot n_3 & \\
\leq \Theta + (c_3 + c_5 + c_6) \cdot (n_1 + n_2 + n_3) & \text{as } n_1 + c_5 \leq n_1 \cdot c_5 \\
\leq \Theta + c \cdot (n_1 + n_2 + n_3) & \text{as } c_3 + c_5 + c_6 \leq c \\
\leq c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + n_1 + n_2 + n_3 + |T|) & \text{cf above} \\
\leq c \cdot ((|\mathit{rul}^{A'}| + n_1 + n_2 + n_3) \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \\
\leq c \cdot (|\mathit{rul}^{A}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \text{cf below}
\end{array}$$

Here we supposed that $n_1 + c_5 \leq n_1 \cdot c_5$, which is true as $n_1 = (|\Sigma| - 1) \cdot 2^{n+1} \geq 2$, and $c_5 \geq 2$. The number of rules of A is exactly: $|\mathit{rul}^{A}| = |\mathit{rul}^{A'}| + n_1 + n_2 + n_3$, which justifies the last inequality.

Case $T = \ell(T')$**.** Let A' be the automaton built for T'. If $\ell = *$ then we can take $A = A'$. Otherwise, $\ell = a \in \Sigma$. We can build A from A' by adding one event state 0 and one node state 0. The event state 0 is a sink state. When opening the root, A checks whether it is labeled by a. If this is the case, A performs the run of A' until the end. Otherwise, A goes to the sink state 0.

$$\mathit{stat}_{\mathrm{e}}^{A} = \mathit{stat}_{\mathrm{e}}^{A'} \uplus \{\mathit{start}, 0\} \qquad \mathit{init}^{A} = \{\mathit{start}\}$$
$$\mathit{stat}_{\mathrm{n}}^{A} = \mathit{stat}_{\mathrm{n}}^{A'} \uplus \{0\} \qquad \mathit{fin}^{A} = \mathit{fin}^{A'}$$

$$\frac{q_1 \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2 \in \mathit{rul}^{A'} \qquad q_1 \in \mathit{init}^{A'}}{\mathit{start} \xrightarrow{\mathit{op}\ (a,v):\gamma} q_2} \quad \text{opening an } a\text{-root}$$

$$\frac{b \in \Sigma - \{a\} \qquad v \subseteq \mathcal{V}_n}{\mathit{start} \xrightarrow{\mathit{op}\ (b,v):0} 0} \quad \text{opening a } b\text{-root, with } b \neq a$$

$$\frac{b \in \Sigma \qquad q_1 \xrightarrow{\alpha\ (b,v):\gamma} q_2 \in \mathit{rul}^{A'}}{q_1 \xrightarrow{\alpha\ (b,v):\gamma} q_2} \quad \text{test } T'$$

$$\frac{b \in \Sigma \qquad v \subseteq \mathcal{V}_n}{0 \xrightarrow{\alpha\ (b,v):0} 0} \quad \text{sink state } 0 \text{ is universal}$$

The number of states of A is:

$$\begin{aligned} |\mathit{stat}_{\mathrm{e}}^{A}| = |\mathit{stat}_{\mathrm{e}}^{A'}| + 2 &\leq (3 \cdot |T'|)^{\mathit{width}(T')} + 2 = (3 \cdot |T'|)^{\mathit{width}(T)} + 2 \\ &\leq (3 \cdot |T|)^{\mathit{width}(T)} \\ |\mathit{stat}_{\mathrm{n}}^{A}| = |\mathit{stat}_{\mathrm{n}}^{A'}| + 1 &\leq (3 \cdot |T'|)^{\mathit{width}(T')} + 1 = (3 \cdot |T'|)^{\mathit{width}(T)} + 1 \\ &\leq (3 \cdot |T|)^{\mathit{width}(T)} \end{aligned}$$

In order to build A, we have to go through the n_1 rules of A' starting from an initial state. This can be done in time $c_3 \cdot n_1$, according to condition (iii). Copying rules of A' has no cost, as we transform A' to A. The n_2 rules for opening a b-node and for the sink state 0 can be built in time n_2. The event, node, initial and final states can be set in time $2 \cdot c_7$ by condition (vii). Let Θ be the time needed to build A' and to set event, node, initial and final states:

$$\begin{array}{lll} \Theta = & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T'|) + 2 \cdot c_7 & \text{by induction hypothesis} \\ \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T'| + 1) & \text{as } 2 \cdot c_7 \leq c \\ \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T')} + |T|) & \text{as } |T| = |T'| + 1 \\ \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \text{as } \mathit{width}(T) = \mathit{width}(T') \end{array}$$

The overall time cost for building A is thus at most:

$$\begin{array}{lll} & \Theta + c_3 \cdot n_1 + n_2 & \\ \leq & \Theta + c_3 \cdot (n_1 + n_2) & \\ \leq & \Theta + c \cdot (n_1 + n_2) & \text{as } c_3 \leq c \\ \leq & c \cdot (|\mathit{rul}^{A'}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + n_1 + n_2 + |T|) & \text{cf above} \\ \leq & c \cdot ((|\mathit{rul}^{A'}| + n_1 + n_2) \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \\ \leq & c \cdot (|\mathit{rul}^{A}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \text{cf below} \end{array}$$

We have $|\mathit{rul}^{A}| = |\mathit{rul}^{A'}| + n_1 + n_2$, so the last inequality is true.

Case $T = x$**.** Suppose that the root of the tree $t \tilde{*} \nu$ is labeled by (a, v). Then the automaton A only needs to check that $x \in v$. We can do it using only two event states (as A must be pseudo-complete).

$$\begin{array}{ll} \mathit{stat}_{\mathrm{e}}^{A} = \{0, 1\} & \mathit{init}^{A} = \{0\} \\ \mathit{stat}_{\mathrm{n}}^{A} = \{0\} & \mathit{fin}^{A} = \{1\} \end{array}$$

$$\frac{q \in \{0,1\} \qquad a \in \Sigma \qquad v \subseteq \mathcal{V}_n}{q \xrightarrow{\mathit{op}\ (a,v):0} 0} \qquad \text{at opening, go to } 0$$

$$\frac{q \in \{0,1\} \qquad a \in \Sigma \qquad v \subseteq \mathcal{V}_n \qquad x \in v}{q \xrightarrow{\mathit{cl}\ (a,v):0} 1} \qquad \text{at closing, go to } 1 \text{ if } x \in v$$

$$\frac{q \in \{0,1\} \qquad a \in \Sigma \qquad v \subseteq \mathcal{V}_n \qquad x \notin v}{q \xrightarrow{\mathit{cl}\ (a,v):0} 0} \qquad \text{at closing, go to } 0 \text{ if } x \notin v$$

A is deterministic and pseudo-complete, and the correctness is immediate. The number of states verifies the desired property:

$$|\mathit{stat}_{\mathrm{e}}^{A}| = 2 \leq (3 \cdot 1)^1 = (3 \cdot |T|)^{\mathit{width}(T)}$$
$$|\mathit{stat}_{\mathrm{n}}^{A}| = 1 \leq (3 \cdot 1)^1 = (3 \cdot |T|)^{\mathit{width}(T)}$$

Building each rule of A is done in time 1, and setting the event, node, initial and final states is done in time $2 \cdot c_7$ by condition (vii), so we can build A in time:

$$\begin{array}{rcll} |\mathit{rul}^A| + 2 \cdot c_7 & \leq & |\mathit{rul}^A| + c & \text{as } 2 \cdot c_7 \leq c \\ & \leq & c \cdot (|\mathit{rul}^A| + 1) & \\ & \leq & c \cdot (|\mathit{rul}^A| + |T|) & \text{as } |T| = 1 \\ & \leq & c \cdot (|\mathit{rul}^A| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|) & \end{array}$$

This completes the proof of Lemma 14. □

In the sequel we extend the definition of canonical trees $t * \tau \in \mathcal{T}_{\Sigma \times 2^{\mathcal{V}_n}}$ built from a tree t and tuple $\tau \in \mathit{nod}(t)^n$. We define canonical trees $t * \mu$ from t and an assignment $\mu : \mathcal{V}_n \to \mathit{nod}(t)$ in the natural way: $t * \mu = t * (\mu(x_1), \ldots, \mu(x_n))$.

Theorem 7. *Let k and n be fixed, and let assume that $|\Sigma| \geq 2$. Given an expression $(T(x_1, \ldots, x_n), B)$ of k-Downward XPath, a pseudo-complete dSTA A over signature $\Sigma \times 2^{\mathcal{V}_n}$ can be computed in polynomial time $O(|T|^{2k} \cdot 30^k \cdot |\Sigma|^{k+1} \cdot 6^n)$ such that for every tree $t \in L(B)$ and μ: $\mathcal{V}_n \to \mathit{nod}(t)$:*

$$t * \mu \in L(A) \qquad \textit{iff} \qquad \epsilon \in [\![T]\!]_{t,\mu}$$

Proof. Let A_0 be the pseudo-complete dSTA obtained for T in time $c \cdot (|\mathit{rul}^{A_0}| \cdot (5 \cdot |\Sigma|)^{\mathit{width}(T)} + |T|)$ by Lemma 14. By condition 1, $\mathit{width}(T) \leq k$, and we have $|\mathit{stat}_{\mathrm{e}}^{A_0}| \leq (3 \cdot |T|)^k$ and $|\mathit{stat}_{\mathrm{n}}^{A_0}| \leq (3 \cdot |T|)^k$. As A_0 is a deterministic STA over alphabet $\Sigma \times 2^{\mathcal{V}_n}$, $|\mathit{rul}^{A_0}|$ is in $O(|\mathit{stat}_{\mathrm{e}}^{A_0}| \cdot |\mathit{stat}_{\mathrm{n}}^{A_0}| \cdot |\Sigma| \cdot 2^n)$, so A_0 can be computed in time $O((3 \cdot |T|)^{2k} \cdot 2^n \cdot 5^k \cdot |\Sigma|^{k+1})$.

To obtain A from A_0, it suffices to intersect A_0 with the dSTA C, that recognizes canonical trees, i.e. trees over signature $\Sigma \times 2^{\mathcal{V}_n}$ where every variable of $\mathcal{V}_n$ appears exactly once. We propose the following construction for C, that simply collects read variables at opening time:

$$\mathit{stat}_{\mathrm{e}}^{C} = 2^{\mathcal{V}_n} \qquad \mathit{init}^{C} = \{\emptyset\} \qquad \mathit{fin}^{C} = \{\mathcal{V}_n\} \qquad \mathit{stat}_{\mathrm{n}}^{C} = \{\_\}$$

$$\frac{a \in \Sigma \qquad v, v' \subseteq \mathcal{V}_n \qquad v \cap v' = \emptyset}{v \xrightarrow{\mathit{op}\ (a,v'):\_} v \cup v' \in \mathit{rul}^{C}} \qquad \frac{a \in \Sigma \qquad v' \subseteq v \subseteq \mathcal{V}_n}{v \xrightarrow{\mathit{cl}\ (a,v'):\_} v \in \mathit{rul}^{C}}$$

We can build C in time $O(|\Sigma| \cdot 3^n)$: For opening rules, choosing v and v' consists in determining for each variable $x \in \mathcal{V}_n$ whether $x \in v - v'$, $x \in v' - v$ or $x \notin v \cup v'$. Similarly, for closing rules, we have to choose whether $x \in v - v'$, $x \in v'$, or $x \notin v \cup v'$.

Let $A = A_0 \cap C$. A accepts canonical trees $t * \mu$ where μ: $\mathcal{V}_n \to \mathit{nod}(t)$. For such a tree t and assignment μ, we know by definition of operator $\tilde{*}$ that $t * \mu \in L(A) \Leftrightarrow t\tilde{*}(\mu^{-1}) \in L(A_0)$. From the definition of A_0, $t\tilde{*}(\mu^{-1}) \in L(A_0) \Leftrightarrow \epsilon \in [\![T]\!]_{t,\mu}$.

The time for building A is the time for building A_0 and C, and the cost of intersecting them. Building A_0 is in time $O((3 \cdot |T|)^{2k} \cdot 2^n \cdot 5^k \cdot |\Sigma|^{k+1})$, and building C in time $O(|\Sigma| \cdot 3^n)$. For the intersection of A_0 and C, we have $|\mathit{rul}^{A_0}|$ in $O((3 \cdot |T|)^{2k} \cdot |\Sigma| \cdot 2^n)$, and $|\mathit{rul}^{C}|$ in $O(|\Sigma| \cdot 3^n)$, so their intersection is in time $O(|\Sigma|^2 \cdot 6^n \cdot (3 \cdot |T|)^{2k})$. Hence the total time for building A is in $O((3 \cdot |T|)^{2k} \cdot (2^n \cdot 5^k \cdot |\Sigma|^{k+1} + |\Sigma|^2 \cdot 6^n))$, which is also $O((3 \cdot |T|)^{2k} \cdot |\Sigma|^{k+1} \cdot 5^k \cdot 6^n)$, and thus $O(|T|^{2k} \cdot |\Sigma|^{k+1} \cdot 30^k \cdot 6^n)$. □

6.2.5 k-Downward XPath is m-streamable for every $m \in \mathbb{N}_0$

Theorem 8. *For every fixed $k, n \geq 0$, the query language k-Downward XPath restricted to n-ary queries is m-streamable for all $m \in \mathbb{N}_0$.*

Proof. Let $(T(x_1, \ldots, x_n), B)$ be an expression of k-Downward XPath, which consists of a filter term T with n variables $x_1, \ldots, x_n$ and a dSTA B over Σ. Let Q be the n-ary query defined by $T(x_1, \ldots, x_n)$, with the schema $L(B)$. Let $\mathcal{A}(Q)$ be the algorithm that first applies the algorithm of Section 6.2.4 in order to translate $T(x_1, \ldots, x_n)$ to a pseudo-complete dSTA A with signature $\Sigma \times 2^{\mathcal{V}_n}$ in PTIME, completes it (also in PTIME for fixed n) and then applies the PTIME precomputation of the query answering algorithm of Chapter 5, to build an SRAM $\mathcal{M}$ computing Q. Let p_0 be a polynomial bounding the time of these steps.

The algorithm of Chapter 5 has the following costs per step (Theorem 5): $O((c+i) \cdot |A|^2 \cdot |B|^2)$ in time and $O(c \cdot d \cdot |A| \cdot |B|)$ in space, where $c = \mathit{concur}_Q(t)$, $i = \mathit{simult\_safe}_Q(t)$ and $d = \mathit{depth}(t)$. Let $m \in \mathbb{N}_0$, and suppose that $c \leq m$. Then, as $i \leq 2^n \cdot c$, $|A|$ being in $O(|T|^{2k})$, and d being bounded by restriction 5, there are polynomials p_1 and p_2 such that for every event of every tree $t \in L(B)$, $\mathit{Space}(\mathcal{A}, t) \leq p_1(|T| \cdot |B|)$ and $\mathit{Time}(\mathcal{A}, t) \leq p_2(|T| \cdot |B|)$. Thus, the query class is m-streamable for p_0, p_1 and p_2. □

The concurrency of k-Downward XPath expressions is not always bounded, however, so that streamability fails by Proposition 9. Even though m-streamable for all $m \in \mathbb{N}_0$, we can define queries with unbounded concurrency in k-Downward XPath. For binary queries, counter examples are easy to construct.

For instance, the query $/(\mathit{and}(\mathit{ch}(a(x)), ch(b(y))))$ selects all (a, b) pairs in trees $c(a, a, a, \ldots, b)$ but nothing in trees $c(a, a, a, \ldots, a)$. This shows that an unbounded number of partial candidates may be alive, where only the first component is instantiated to some a-node. The previous example can also be adapted to the monadic case, with the expression $/(\mathit{and}(\mathit{ch}(a(x)), ch(b(\mathit{true}))))$. With the same trees, we also prove that this query has unbounded concurrency.

In many practical use cases, m-streamability suffices to ensure the existence of efficient algorithms. Consider for instance a bibliography file, where every child of the root describes a book. Consider also the query Q that looks for co-authors of a given author a. The concurrency of Q may be unbounded, as we can read an unbounded number of authors under a book, before reaching an a-node. However, in practice, the number of co-authors is low, and queries in m-streamable query classes, where m is greater than the maximal number of co-authors, can be processed with polynomial per-step space and time cost.

6.3 Beyond k-Downward XPath: Prospective Ideas

In this section we propose two extensions of k-Downward XPath. The first one limits the concurrency, in order to obtain an ∞-streamable fragment of XPath. The second extension adds horizontal axes *ns* and *ns**. **This section intends to provide prospective ideas for future work. Most results are not proved and should be considered as conjectures.**

6.3.1 ∞-Streamable Fragments of Forward XPath

In Theorem 8, we proved that k-Downward XPath is m-streamable for all $m \in \mathbb{N}_0$. As previously mentioned, it is however not ∞-streamable. However, restricting k-Downward XPath to queries of polynomially bounded concurrency would be sufficient.

Theorem 9. *Every fragment of k-Downward XPath having polynomially bounded concurrency is ∞-streamable, for every $k \in \mathbb{N}$.*

Proof. By Proposition 8 and hypothesis, it suffices to show that there exist polynomials p_0, p_1 and p_2 such that for all $m \in \mathbb{N}_0$, k-Downward XPath is m-streamable with p_0, p_1, p_2. We need to prove that polynomials p_0, p_1 and p_2 used in the proof of Theorem 8 are independent from m. This is obviously the case for p_0. The concurrency is polynomially bounded (by hypothesis), so there exists a polynomial p such that $\mathit{concur}_{Q_e}(t) \leq p(|e|)$ for all $t \in \mathcal{T}_\Sigma$ and all expressions e in k-Downward XPath. If $e = (T(x_1, \ldots, x_n), B)$, then by Theorem 7, $T(x_1, \ldots, x_n)$ can be converted in PTIME into a dSTA A recognizing $L_{T(x_1,\ldots,x_n)}$.

Hence there exists a polynomial p' such that $|A| \leq p'(|T(x_1, \ldots, x_n)|)$, and we can find polynomials p_1 and p_2 such that for every expression $(T(x_1, \ldots, x_n), B)$ of k-Downward XPath, $\mathit{Space}(\mathcal{A}, t) \leq p_1(|T(x_1, \ldots, x_n)| \cdot |B|)$ and $\mathit{Time}(\mathcal{A}, t) \leq p_2(|T(x_1, \ldots, x_n)| \cdot |B|)$. □

Thanks to downward axes and guards on ch^* axes, every branch of k-Downward XPath queries only has at most one match at a time. This is however not sufficient to bound concurrency. We propose two additional sets of conditions, in order to obtain a fragment of -Downward XPath with bounded concurrency.

Variables below negation In the sequel we call *positions* the set of nodes of a term T. A *negative* position is a position with an odd number of strict ancestors labeled by *not*. An *or* -position is a negative position labeled by *and*.

Variables in negative positions raise trouble. Consider for instance the query $/(\mathit{and}(\mathit{not}(x), \mathit{ch}^*(a(\mathit{true}))))$ which selects all nodes x that are not the root, if the tree contains an a-node. This query has unbounded concurrency. The problem is variable x which occurs in negative position, so that it does not have to match the current position. We have to forbid variables in negative positions all over (condition 7 below). Note that the selecting position in CoreXPath 1.0 expressions is always positive, so this restriction is quite natural.

Variables in disjunctions are a further source of trouble. For instance, consider the query defined by $/(\mathit{or}(\mathit{ch}^*(c(\mathit{and}(x, \mathit{ch}^*(a(\mathit{true}))))), \mathit{ch}^*(b(\mathit{true}))))$ which selects all nodes in tree $a(a(\ldots(a(b))))$, where the second branch becomes true independently of the value of variable x in the first branch. A streaming algorithm can decide selection only at the end when opening the b-leaf. Thus this query has unbounded concurrency. This query can be expressed in our dialect of Forward XPath, by using conjunction and negation. We need to impose that all choices of *or*-positions contain the same variables (condition 6).

Variables below axes or label tests in negative positions raise trouble. Consider for instance the query: for all a-nodes there exists a b-child which is selected, i.e., $/(\mathit{not}(\mathit{ch}^*(a(\mathit{not}(\mathit{ch}(b(x)))))))$. This query selects all c-nodes in tree $c(c, \ldots, c)$ but not in $c(c, \ldots, c, a(b))$ where it selects the b-node. Thus, none of the c-nodes is safe for selection before the end, i.e., the concurrency of the query is unbounded. In order to avoid this, we have to forbid variables below occurrences of axes (resp. label tests) in negative position (condition 8). This again is satisfied by all paths of CoreXPath 1.0.

Variables below conjunction Queries using conjunctions may have high concurrency. Consider for instance the query defined by the expression $/(\mathit{and}(\mathit{ch}(a(x)), \mathit{ch}(b(\mathit{true}))))$, that selects all a-children of the root, if the root has a b-child. It selects all a-nodes on tree $b(a, \ldots, a, b)$ but nothing on tree

$b(a, \ldots, a)$, and thus has unbounded concurrency. This query implicitly tests among siblings of nodes. We can avoid this effect by forbidding axes between *and*-positions and variables, as expressed by condition 9 below.

Weak k-Downward XPath Let $\mathcal{V}(T)$ be the set of variables used in a term T. The query language *Weak k-Downward XPath* provides all pairs (T, B) of k-Downward XPath that satisfy the following further restrictions:

6. all *or*-position p of T with choices $T_1, \ldots, T_n$ satisfy $\mathcal{V}(T_1) = \ldots = \mathcal{V}(T_n)$, i.e. use the same variables.

7. variables appear in positive positions only, i.e., if $\mathit{lab}_x^T(p)$ then there is an even number of *not*-labeled positions above p.

8. on the branch of a position p labeled x, there is no negative position labeled by an axis d or a label test ℓ.

9. no position labeled by an axis d can have both a descendant, labeled by a variable x, and an ancestor, labeled by *and*.

We conjecture that monadic queries in Weak k-Downward XPath have concurrency at most 2, and thus that Monadic Weak k-Downward XPath is ∞-streamable.

6.3.2 Adding Horizontal Axes

In this section we propose some ideas for dealing with horizontal axes *ns* and *ns**. The major difference with downward axes is that selection of nodes (or their validity w.r.t. to a match) cannot always be decided at closing time.

Deciding at Last Siblings A solution is to postpone this decision to the closing time of the parent node. Indeed, suppose that we want to know whether a node $\pi \in \mathit{nod}(t)$ verifies a filter term T, i.e., whether $\pi \in [\![T]\!]_{t,\mu}$. Then, as we only use axes $\mathcal{D} = \{\mathit{ch}, \mathit{ch}^*, \mathit{ns}, \mathit{ns}^*\}$, the validity of $\pi \in [\![T]\!]_{t,\mu}$ can be decided when all right-siblings of π and their descendants have been seen. The earliest time point where we know that all this region has been read is at closing the parent of π.

In order to maintain a PTIME translation to dSTAs, we need to still have at most one match to compute at a time. This implies some updates in conditions 1 to 5. For instance, label guards must be imposed for both *ch** and *ns**, and if such a guard symbol a is in T, then a-nodes are forbidden among right-siblings of a-nodes, and their descendants.

k-Forward XPath We define an extension of k-Downward XPath with axes $\mathcal{D} = \{ch, ch^*, ns, ns^*\}$. For $k \in \mathbb{N}$, *k-Forward XPath* is the query class containing all pairs $(T(x_1, \ldots, x_n), B)$ of terms T with a sequence of variables $(x_1, \ldots, x_n)$ and dSTAs B over alphabet Σ verifying the following conditions:

1. the width of T is bounded by k, i.e., T has at most k leaves.
2. the term T starts at the root, i.e., T matches some term $/(T')$.
3. if $d(T')$ is a subterm of T, with $d \in \mathcal{D}$, then T' matches some term $a(T'')$, with $a \in \Sigma$.
4. if $d(a(T''))$ is a subterm of T, where $d \in \mathcal{D}$, then:

$$\forall t \in L(B).\, \forall \pi_a, \pi'_a \in lab_a(t).\, \nexists \pi \in nod(t).\, \begin{cases} & \pi_a \neq \pi'_a \\ \wedge & (\pi_a, \pi) \in (ns^*)^t \\ \wedge & (\pi, \pi'_a) \in (ch^*)^t \end{cases}$$

5. the depth of the valid trees $t \in L(B)$ is bounded by some constant.

All conditions are identical to those of k-Downward XPath except conditions 3 and 4. Condition 3 imposes a label guard below every axis position. If such a guard a appears in T, then condition 4 forbids a-nodes among descendants of right-siblings of another a-node in $t \in L(B)$. Hence, before testing a new match for the a-position, we can decide the validity of the previous match for this a-position.

We conjecture that the algorithm translating k-Downward XPath expressions into dSTAs in PTIME can be easily adapted to k-Forward XPath. The only treatments to change are those for axes, i.e., $T = d(a(T'))$. Instead of running the automaton for T' until closing the a-node π, we have to run it until closing the parent node of π. If this holds, this would also mean that k-Forward XPath is m-streamable for all $m \in \mathbb{N}_0$.

Let Weak k-Forward XPath be the fragment of k-Forward XPath with the additional restrictions 6 to 9 of Weak k-Downward XPath. We also conjecture that Weak k-Forward XPath is ∞-streamable. Moreover, membership to k-Forward XPath and Weak k-Forward XPath can be decided in PTIME.

Discussion on Improvements The restrictions of k-Forward XPath are quite strong. Consider for instance the query Q_1 defined by the expression $/(ch^*(a(ch(b(x)))))$, and the query Q_2 defined by $/(ch^*(a(ns^*(b(x)))))$. Query Q_1 selects all b-nodes having an a-node as parent, whereas Q_2 selects all b-nodes having an a-node as previous sibling. In k-Forward XPath, for both queries, no

a-node can appear among next-siblings of a, and their descendants. For Q_2, forbidding a-nodes among right-siblings of a-nodes avoids unbounded concurrency, as for instance in tree $c(a, \ldots, a, b)$. Nevertheless, this is useless for Q_1, as the subterm $\mathit{ch}(b(x))$ below $\mathit{ch}^*(a)$ looks for matches only in descendants of a-nodes.

This would justify to introduce a notion of *scope*, where $\mathit{scope}^t_\pi(T)$ would contain the region from which the truth value of $\pi \in [\![T]\!]_{t,\mu}$ depends. In the previous example, $\mathit{scope}^t_\pi(\mathit{ch}(b(x)))$ would contain children of π, whereas $\mathit{scope}^t_\pi(\mathit{ns}^*(b(x)))$ would contain all right-siblings of π. Hence, instead of forbidding a-node in right-siblings and their descendants when $d(a(T'))$ appears in T, we would forbid a-nodes only in $\mathit{scope}^t_\pi(T')$, for all $t \in L(B)$ and $\pi \in \mathit{lab}_a(t)$.

6.4 Conclusion

After non-streamability results on Forward XPath in Chapter 3, we presented in the present chapter the hierarchy k-Downward XPath (for $k \in \mathbb{N}$) of query classes enjoying streamability properties. To prove these properties, we provided a translation to dSTAs in PTIME. We also proposed some insights for ∞-streamable extensions, and for extensions allowing rightward moves. We discuss in the following some further possible improvements and open issues related to these fragments and their translations.

The first question is whether k-Forward XPath can be enlarged, while remaining m-streamable for all $m \in \mathbb{N}_0$. In our translation we excluded one forward axis: the *foll* axis. We conjecture that adding this axis is not a problem in the translation, as the end of scope for this axis is always when the root is closed, which can be easily detected. However strong restrictions on valid trees will have to be added, as the presence of a step $\mathit{foll}::a$ will impose that there is at most one a-node per valid trees. About extending k-Forward XPath, an open question is the definition of a necessary and sufficient criterion on Forward XPath fragments, that ensures PTIME translation to dSTAs.

Concerning Weak k-Forward XPath, we conjectured that restrictions 6 to 8 imply bounded concurrency, whereas polynomially bounded concurrency would be sufficient for being ∞-streamable. This leads to an open question: can we take weaker restrictions and remain polynomially bounded?

One may also want to improve the proposed translation for k-Downward XPath, in order to infer assertions at their earliest position, and thus get an automaton A being earliest, like $E(A)$ in Chapter 5. In order to obtain this property, the algorithm has to take the schema into account, as it will sometimes have to infer assertions before their corresponding ends of scope, as some subterm of T might be unsatisfiable or always satisfied for every continuation of $t * \mu$ beyond the current event. It is also open whether this could be done efficiently.

Another question is whether we could get better query answering algorithms without translating the XPath expressions to dSTAs, but rather working directly with the XPath expressions. We are not optimistic about such improvements, as dSTAs are close to the implementation level of XML streaming algorithms, and in our translation, only relevant information is stored into the states of the automaton.

Chapter 7

Deciding Bounded Delay and Concurrency

Contents

7.1 Introduction

The class $\mathcal{Q}^d_{\text{dSTAs}}$ of queries defined by dSTAs where valid trees have depth at most d is m-streamable, for all $m \in \mathbb{N}_0$. $\mathcal{Q}^d_{\text{dSTAs}}$ is however not ∞-streamable, as it contains queries of unbounded concurrency defined by small dSTAs. The m-streamability of $\mathcal{Q}^d_{\text{dSTAs}}$ means that queries in this class can be efficiently evaluated, when the concurrency of queries w.r.t. input trees is smaller than m. Hence, bounding the concurrency of queries w.r.t. all valid trees ensures efficient evaluation in streaming mode. Let $\mathcal{Q}^{d,c}_{\text{dSTAs}}$ be the subclass of $\mathcal{Q}^d_{\text{dSTAs}}$ containing queries having concurrency at most c on all valid trees. By Proposition 8, $\mathcal{Q}^{d,c}_{\text{dSTAs}}$ is ∞-streamable.

In this chapter, we prove that it can be decided in polynomial time whether a query defined by a dSTA has bounded concurrency on all valid trees, and whether for a given k, the concurrency is bounded by k. This provides an efficient procedure for deciding whether a query belongs to $\mathcal{Q}^{d,c}_{\text{dSTAs}}$.

To establish that boundedness for concurrency is decidable in PTIME, we use automata techniques. We start with the case of queries over words, defined by standard deterministic word automata. Bounded (and k-bounded) ambiguity of word automata is known to be decidable in PTIME, as studied for instance by Stearns and Hunt [SH85], Weber and Seidl [WS86], or more recently by Allauzen et al. [AMR08]. We transform automata defining queries to non-deterministic automata, whose ambiguity is exactly the concurrency of queries. Hence, we lift the decision problem from bounded concurrency to bounded ambiguity.

For trees, however, this method cannot be used directly. We choose to use recognizable relations, as studied by Tison for ranked trees [Tis90, CDG$^+$07], and recently investigated by Benedikt et al. [BL02, BLN07] for unranked trees. A relation between trees is recognizable if the set of overlays of tuples in this relation is recognized by some tree automaton. Concurrency of queries defined by automata can be expressed by recognizable relations. We show how to define the relation capturing concurrency by first-order formulas with tree-valued variables, from the automaton defining the query. Our reduction is in PTIME if we assume determinism, since we only use a restricted class of first-order formulas in prenex normal form, where all quantifiers are existential. Note that quantification over trees (instead of nodes of trees in MSO) allows us to express in a direct manner properties of queries to be checked on *all continuations* of the stream.

In order to obtain our PTIME decision procedure, we prove that for fixed k, bounded and k-bounded valuedness of binary recognizable relations can be decided in polynomial time even when the automaton defining the relation is non-

deterministic (when k is variable, it becomes EXPTIME-complete). The valuedness of a binary relation R is the maximal number of trees t_2 that are in relation with the same tree t_1 in the first component, i.e. such that $(t_1, t_2) \in R$. For bounded valuedness, we reduce the problem to the bounded valuedness of tree transducers, studied by Seidl in [Sei92]. For k-bounded valuedness, we use the equivalence between operations on relations and operations on automata.

In [BL02, BLN07], Benedikt et al. define two extension operators (downward and rightward) plus an operator checking that a relation is a relabeling (i.e. relates trees with the same shape). They prove that a relation is recognizable if and only if it can be expressed by an FO formula, using these operators as predicates. Compared to this work by Benedikt et al., our results on valuedness are new.

In addition to concurrency, we are interested in the maximal *delay* of a query, for which we obtain similar decidability results. For monadic queries, the delay is the number of events between reading a selected node, and the earliest time point from which its selection can be safely decided, i.e., from which any continuation of the stream will select it. For n-ary queries, we start counting when the tuple becomes complete (as it cannot be output before).

Bounded delay is interesting for two reasons. First, the delay of a query measures quality of service, whereas the concurrency measures the memory requirements. It bounds the waiting time for selection, in terms of number of events. Second, bounded delay implies bounded concurrency, for monadic queries. Moreover, the delay of a query is easier to characterize than its concurrency. Hence, for query over words, we give a direct procedure for computing the delay. For queries over trees, bounded delay can be decided in PTIME when the arity n of queries is variable, whereas we have to fix it for deciding bounded concurrency in PTIME.

For n-ary queries, delay and concurrency are incomparable. A query with bounded concurrency but unbounded delay is easy to find, for instance the query that selects the root if its last child is labeled by a. Its concurrency is bounded, as only the root node is alive, but the delay is the number of events between opening the root and closing its last child, and thus unbounded. On the contrary, we can build queries with bounded delay but unbounded concurrency. This is due to the fact that concurrency takes partial tuples into account, but the delay does not. Hence we can build queries that generate a lot of partial candidates, but for which the answer tuples can be output immediately once they get complete. This is for instance the case, for the query that selects all pairs of nodes. It requires to buffer all partial tuples containing previously opened nodes in one component. Once a new node is read, we can complete all these partial tuples with this node, and output the resulting tuples immediately, without delay.

7.2 Delay and Concurrency for Words and Trees

We generalize the earliest query answering definitions of Section 3.2 to both words and trees. We define the notion of delay, and generalize concurrency to words and trees.

7.2.1 EQA for Words and Trees

We consider the cases of words and trees in simultaneously, where either $\mathcal{S} = \Sigma^*$ is the set of all words or $\mathcal{S} = \mathcal{T}_\Sigma$ the set of all unranked tree over Σ.

We consider words as relational structures, as introduced in Section 2.1.2. A word $w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*$ has domain $\mathit{dom}(w) = \{1, \ldots, k\}$, and by analogy with trees, we define its set of events by: $\mathit{eve}(w) = \{0, \ldots, k\}$. Given a word $w \in \Sigma^*$, we write $\mathit{dom}_\eta(w) = \{1, \ldots, \eta\}$ for the set of positions of w visited before the event η, and $\mathit{dom}^\bullet_\eta(w) = \mathit{dom}_\eta(w) \cup \{\bullet\}$.

Let Q be an n-ary query in structures $\mathcal{S}$, $s \in \mathcal{S}$ a structure, and $\eta \in \mathit{eve}(s)$ an event of s. A *complete candidate* until event η is a tuple $\tau \in \mathit{dom}_\eta(s)^n$. Given two structures $s_1, s_2 \in \mathcal{S}$ and an event $\eta \in \mathit{eve}(s_1) \cup \mathit{eve}(s_2)$, we say that the prefixes of the linearizations of s_1 and s_2 until η coincide, if:

$$\mathit{equal}_\eta(s_1, s_2) \Leftrightarrow \begin{cases} \mathit{dom}_\eta(s_1) = \mathit{dom}_\eta(s_2) \wedge \\ \forall a \in \Sigma.\ \forall \pi \in \mathit{dom}_\eta(s_1).\ (\mathit{lab}^{s_1}_a(\pi) \Leftrightarrow \mathit{lab}^{s_2}_a(\pi)) \end{cases}$$

Definitions of sufficient events for selection (resp. rejection) are easily lifted to arbitrary structures. We write $\mathit{compl}(\tau, s, \eta)$ for the set of complete candidates, in which all unknown components of τ have been instantiated with elements $\pi \in \mathit{dom}(s) - \mathit{dom}_\eta(s)$.

$$(\tau, \eta) \in \mathit{sel}_Q(s) \;\Leftrightarrow\; \tau \in \mathit{dom}_\eta(s)^n \wedge \forall s' \in \mathit{dom}(Q).\ \mathit{equal}_\eta(s, s') \Rightarrow \tau \in Q(s')$$

$$(\tau, \eta) \in \mathit{rej}_Q(s) \Leftrightarrow \begin{cases} \tau \in \mathit{dom}^\bullet_\eta(s)^n \wedge \\ \forall s' \in \mathit{dom}(Q).\ \mathit{equal}_\eta(s, s') \Rightarrow \\ \qquad \forall \tau' \in \mathit{compl}(\tau, s', \eta).\ \tau' \notin Q(s') \end{cases}$$

Alive candidates at η if τ are those being neither rejected nor selected at η.

$$(\tau, \eta) \in \mathit{alive}_Q(s) \Leftrightarrow \tau \in \mathit{dom}^\bullet_\eta(s)^n \text{ and } (\tau, \eta) \notin \mathit{rej}_Q(s) \text{ and } (\tau, \eta) \notin \mathit{sel}_Q(s)$$

We introduce the concurrency at an event η, which is more fine-grained than the global concurrency defined in Section 3.2.3.

$$\mathit{concur}_Q(s, \eta) = |\{\tau \in \mathit{dom}^\bullet_\eta(s)^n \;\mid\; (\tau, \eta) \in \mathit{alive}_Q(s)\}|$$

With this definition, we obtain the following equivalence with our previous notion of concurrency: $\mathit{concur}_Q(s) = \max_{\eta \in \mathit{eve}(s)} \mathit{concur}_Q(s, \eta)$.

7.2.2 Delay

We formally introduce the notion of *delay* in our query answering framework, for both words and trees. For monadic queries, it is the number of events between a node and the earliest event for its selection. Given a structure s (a word or a tree), let

$$\mathit{latest}((\pi_1, \ldots, \pi_n)) = \min\{\eta \in \mathit{eve}(s) \mid \pi_1, \ldots, \pi_n \in \mathit{dom}_\eta(s)\}$$

be the minimal event, where all elements of the tuple have been visited.

Definition 12 (Delay)**.** *The* delay *of an n-ary query Q for a tuple $\tau \in \mathit{dom}(s)$ is the number of events η following $\mathit{latest}(\tau)$ such that η is insufficient for selection, i.e. $(\tau, \eta) \notin \mathit{sel}_Q(s)$.*

$$\mathit{delay}_Q(s, \tau) = |\{\eta \in \mathit{eve}(s) \mid \mathit{latest}(\tau) \preceq \eta,\ (\tau, \eta) \notin \mathit{sel}_Q(s)\}|$$

*A query Q has k-*bounded delay *if $\mathit{delay}_Q(s, \tau) \leq k$ for all $s \in \mathit{dom}(Q)$ and $\tau \in Q(w)$. It has bounded delay if it has k-bounded delay for some $k \geq 0$.*

Having bounded delay means that every EQA algorithm will output selected tuples a constant time after completion. This is a guarantee on the quality of service.

7.2.3 Link to Concurrency

For monadic queries, some links exist between concurrency and delay.

Lemma 15. *For all monadic queries Q, structures $s \in \mathit{dom}(Q)$, and events $\eta \in \mathit{eve}(s)$:*

$$\mathit{concur}_Q(s, \eta) \leq \sup_{s' \in \mathit{dom}(Q), \tau \in Q(s')} \mathit{delay}_Q(s', \tau) + 1$$

The lemma fails for queries of higher arities, where the delay between the tuple components may be unbounded even though the delay of selection of complete tuples is bounded. In this case, the set of alive partial tuples may grow without bound, even though the set of alive complete tuples is bounded. For instance consider the query Q with $Q(t) = \mathit{nod}(t)^2$ for all trees $t \in \mathcal{T}_\Sigma$. This query has delay 0, since every pair of nodes can be selected immediately, once the its last component has been visited. Nevertheless, all partial tuples $(\pi, \bullet)$ with $\pi \in \mathit{dom}_\eta(t)$ are alive at all events η, so that the concurrency of this query is not bounded.

Proof. Let $s' \in \mathcal{S}$ and $k \in \mathbb{N}_0 \cup \{\infty\}$. In the case of words (where $\mathcal{S} = \Sigma^*$), we define $\mathit{dom}^k_\eta(s')$ by $\{\pi' \mid \eta - k \leq \pi' \leq \eta\}$, and in the case of trees (where $\mathcal{S} = \mathcal{T}_\Sigma$), we define $\mathit{dom}^k_\eta(s')$ as $\{\pi' \mid \mathit{pr}^k(\eta) \leq (\mathit{op}, \pi') \leq \eta\}$.

Let Q be a monadic query. Let $d = sup_{s' \in dom(Q), \tau \in Q(s')} delay_Q(s', \tau)$ be the number in the lemma, and $s \in dom(Q)$ be a structure with event $\eta \in eve(s)$. We claim for all $\pi \in dom(s)$ that:

$$\pi \notin dom^d_\eta(s) \Rightarrow ((\pi), \eta) \notin alive_Q(s)$$

To see this, we first note that if $\pi \notin dom_\eta(s)$ then π is not alive at η. Now let us consider $\pi \in dom_\eta(s) - dom^d_\eta(s)$. We distinguish two cases.

1. In the first case, there exists a continuation $s' \in dom(Q)$ with $equal_\eta(s, s')$ such that $(\pi) \in Q(s')$. This continuation s' satisfies $delay_Q(s', (\pi)) \leq d$, so that $\pi \in dom_\eta(s) - dom^d_\eta(s)$ yields $((\pi), \eta) \in sel(s)$. This contradicts aliveness.

2. Otherwise, all continuations s' of s beyond η satisfy $(\pi) \notin Q(s')$, so that $((\pi), \eta) \in rej(s)$. This equally implies non-aliveness.

This proves the claim, which yields for all partial tuples τ:

$$(\tau, \eta) \in alive_Q(s) \Rightarrow \tau \in dom^d_\eta(s) \cup \{\bullet\}$$

Hence, $concur_Q(s, \eta) \leq d + 1$ by definition of concurrency. □

Proposition 23. *A monadic query with k-bounded delay has $(k+1)$-bounded concurrency.*

Proof. This is an immediate consequence of Lemma 15. □

The converse does not hold. As a counter example, consider the monadic query which selects the first letter of all words whose last letter is a b. This query has concurrency bounded by 1, since the first letter is the only alive candidate before the end, but unbounded delay.

7.3 Bounded Delay and Concurrency for Queries in Words

We consider the case, where queries in words are defined by two deterministic finite automata, that recognize the canonical language of the query and its schema respectively. We obtain PTIME decision procedures for bounded delay and concurrency by reduction to bounded ambiguity of non-deterministic finite automata.

7.3.1 Finite Automata

A *finite automaton* (nFA) over Σ is a tuple $A = (\mathit{stat}, \mathit{init}, \mathit{rul}, \mathit{fin})$ where *init*, *fin* and *stat* are finite sets with $\mathit{init}, \mathit{fin} \subseteq \mathit{stat}$, and $\mathit{rul} \subseteq \mathit{stat}^2 \times (\Sigma \cup \{\epsilon\})$ contains rules that we write as $q \xrightarrow{a} q'$ or $q \xrightarrow{\epsilon} q'$ where $q, q' \in \mathit{stat}$ and $a \in \Sigma$. Whenever necessary, we will index the components of A by A. Let the size of A count all states and rules, i.e. $|A| = |\mathit{stat}_A| + |\mathit{rul}_A|$. We also sometimes use the notation $A[\mathit{init}{=}I]$ (resp. $A[\mathit{fin}{=}I]$) for the automaton obtained from A by setting its initial (resp. final) states to I.

A run of A on a word w is a function $r : \mathit{eve}(w) \to \mathit{stat}_A$ so that $r(0) \in \mathit{init}_A$ and $r(\pi{-}1) \xrightarrow{\epsilon^*} \xrightarrow{a} \xrightarrow{\epsilon^*} r(\pi)$ is justified by *rul* for all $\pi \in \mathit{dom}(w)$ with $a = \mathit{lab}^w(\pi)$. A run is successful if $r(|w|) \in \mathit{fin}_A$. The language $L(A) \subseteq \Sigma^*$ is the set of all words that permit a successful run by A. An nFA is called *productive*, if all its states are used in some successful run. This is the case if all states are reachable from some initial state, and if for all states, some final state can be reached.

An nFA A is *deterministic* or a dFA if it has at most one initial state, no epsilon rules, and for every pair (q, a) there exists at most one rule $q \xrightarrow{a} q' \in \mathit{rul}_A$. Note that for every word w there exists at most one run by a dFA A.

Bounded Ambiguity

We next consider the degree of ambiguity of nFAs A. The *ambiguity* $\mathit{amb}_A(w)$ is the number of successful runs of A on w. Clearly, $\mathit{amb}_A(w) \leq 1$ for all $w \in \Sigma^*$ if A is a dFA. We call the ambiguity of A *k-bounded* if $\mathit{amb}_A(w) \leq k$ for all $w \in \Sigma^*$. It is *bounded*, if it is bounded by some k.

Stearns and Hunt [SH85] (Theorem 4.1) present for fixed $k \in \mathbb{N}$ a PTIME algorithm for deciding k-bounded ambiguity of nFAs. Let us write $p \xrightarrow{w} q$ *by* A if there exists a run of $A[\mathit{init}{=}\{p\}]$ on w that ends in q. Weber and Seidl [WS86] show that an nFA A has unbounded ambiguity iff there exists a word $w \in \Sigma^+$ and distinct states $p \neq q$ such that $p \xrightarrow{w} p$, $p \xrightarrow{w} q$, and $q \xrightarrow{w} q$ by A. This can be tested in $O(|A|^3)$ as shown very recently by [AMR08].

7.3.2 Defining n-ary Queries

As usual, we can define queries by two automata, one for the canonical language and another for the schema. We call an nFA canonical if and only if its language is. Let A be a canonical nFA A with alphabet $\Sigma \times 2^{\mathcal{V}_n}$ and B an nFA with alphabet Σ, such that $w \in L(B)$ for all $w * \tau \in L(A)$. The query $Q_{A,B}$ defined by the pair (A, B) is the unique n-ary query with domain $L(B)$ and canonical language $L(A)$. If $L(B) = \Sigma^+$ then we write Q_A instead of $Q_{A,B}$. Automaton B is needed

in order to distinguish those words, on which the query is not defined, from those, where the query returns the empty set. Note that if $Q_{A,B}(w) \neq \emptyset$ then $w \in L(B)$.

Let the type of a word w with alphabet $\Sigma \times 2^{\mathcal{V}_n}$ be a function $\mathit{type}_w : \mathcal{V}_n \to \mathbb{N}_0$ that counts how many times a variable appears in labels, i.e., for $x \in \mathcal{V}_n$:

$$\mathit{type}_w(x) = |\{\pi \in \mathit{dom}(w) \mid \mathit{lab}^w_{(a,v)}(\pi) \text{ with } x \in v\}|$$

We say that a word w has type $1^{\mathcal{V}_n}$ if $\mathit{type}_w(x) = 1$ for all $x \in \mathcal{V}_n$. All words over $\Sigma \times 2^{\mathcal{V}_n}$ of type $1^{\mathcal{V}_n}$ have the form $w * \tau$, and vice versa. We next show that all states of productive canonical nFAs have unique types. This was already noticed in Lemma 3 of [CLN04]:

Lemma 16. *If A is a productive canonical nFA and $q \in \mathit{stat}_A$ then all words recognized by $A[\mathit{fin} = \{q\}]$ have the same type.*

Proof. Since A is productive, there exists a word $w \in L(A[\mathit{init} = \{q\}])$. Assume that there exist words $w_1, w_2 \in L(A[\mathit{fin} = \{q\}])$ with different types. Hence, the words $w_1 \cdot w$ and $w_2 \cdot w$ must have different types, since $\mathit{type}_{w_1 w} = \mathit{type}_{w_1} + \mathit{type}_w \neq \mathit{type}_{w_2} + \mathit{type}_w = \mathit{type}_{w_2 w}$. This is impossible, though, since $L(A)$ is canonical, so that $\mathit{type}_{w_2 w}(x) = \mathit{type}_{w_1 w}(x) = 1$ for all $x \in \mathcal{V}_n$ □

We can thus define the type of a state q of a productive canonical nFA in a unique manner, by the type of some word w, that is evaluated into this state. $\mathit{type}(q)$ will denote this type. Furthermore, as the automaton is canonical and productive, this type is determined by the set $\{x \in \mathcal{V}_n \mid \mathit{type}_w(x) = 1\}$. So we can identify the type of a state with a subset of $\mathcal{V}_n$.

Consider the query $Q_{\phi(x_1)}$ in words with alphabet $\{a, b\}$, which selects all positions labeled by a or eventually succeeded by an a. In Figure 7.1, we illustrate an automaton for the canonical language of this query graphically. Its states have the following types: $\emptyset$ for q_0 (no variables seen before entering in this state), and $\{x_1\}$ for q_1 and q_2 (x_1 seen before entering in these states).

Query answering for dFAs is the algorithmic problem that receives as input two dFAs A and dFA B defining an n-ary query and a word $w \in L(B)$, and returns as output $Q_{A,B}(w)$. The objective is to find all tuples τ of positions in w such that $w * \tau \in L(A)$. The naive algorithm enumerates all tuples $\tau \in \mathit{dom}(w)^n$ and runs A deterministically on $w * \tau$. This algorithm first resolves the choice of τ nondeterministically, before running the deterministic automaton A.

Determinism for canonical automata will turn out to be essential for PTIME streaming algorithms and decision complexity (e.g. the safety property below). It should be noticed that canonical nFAs can always be determinized without changing the query they define. This would fail when defining queries by selection automata, i.e. nFAs over Σ with a set of selection states as considered in [FGK03, NPTT05].

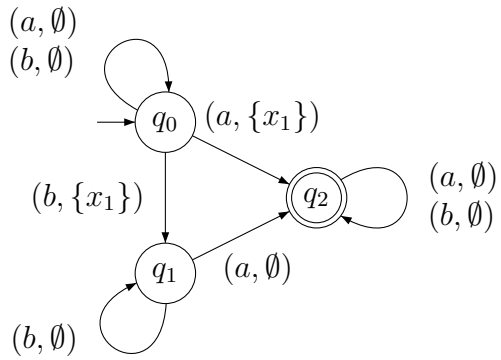


Figure 7.1: A dFA for the canonical language of $Q_{\phi(x_1)}$ where $\phi = \exists x_2.\ (x_1 \leq x_2 \wedge \mathit{lab}_a(x_2))$.

7.3.3 Computing Delays of Queries

We show how to decide whether a query has bounded delay and how to compute this delay in polynomial time. We consider the case with schemas. Schema elimination as proposed in Section 5.4.3 can easily be adapted to queries over relational structures. However, it would require to fix the arity n of the queries, and spoil small polynomials: Given automata A and B defining $Q = Q_{A,B}$, we cannot build an automaton recognizing Q_{sel} or Q_{rej} without a blowup in $O(2^n)$ in the general case, since we have to extend the alphabet of B from Σ to $\Sigma \times 2^{\mathcal{V}_n}$.

Safe States for Selection

For every language $L \subseteq \Sigma^+$ we define a language of annotated words $L \otimes \emptyset$ with alphabet $\Sigma \times 2^{\mathcal{V}_n}$ such that all letters of words in L are annotated by $\emptyset$, i.e., $L \otimes \emptyset = \{(a_1, \emptyset) \cdot \ldots \cdot (a_k, \emptyset) \mid a_1 \cdot \ldots \cdot a_k \in L\}$

Definition 13. *If dFAs A and B define a query then we call a state $(p, q) \in \mathit{stat}_A \times \mathit{stat}_B$* safe for selection *by $Q_{A,B}$ if $L(B[\mathit{init}{=}\{q\}]) \otimes \emptyset \subseteq L(A[\mathit{init}{=}\{p\}])$.*

Figure 7.2 illustrates an automaton for the query that selects all a-nodes that are succeeded by $b \cdot b$. In this example, we assume the universal schema B with a single state, so that A is isomorphic to $P(A, B)$. The types and safety properties of all states are indicated in the figure.

We next show that safe states capture sufficiency for selection. In order to do so, we construct a dFA $P(A, B)$ which runs A and B in parallel. Its alphabet is $\Sigma \times 2^{\mathcal{V}_n}$ as for A, while B has alphabet Σ.

$$\begin{aligned} \mathit{stat}_{P(A,B)} &= \mathit{stat}_A \times \mathit{stat}_B \\ \mathit{init}_{P(A,B)} &= \mathit{init}_A \times \mathit{init}_B \\ \mathit{fin}_{P(A,B)} &= \mathit{fin}_A \times \mathit{fin}_B \end{aligned} \qquad \frac{p \xrightarrow{(a,v)} p' \in \mathit{rul}_A \qquad q \xrightarrow{a} q' \in \mathit{rul}_B}{(p, q) \xrightarrow{(a,v)} (p', q') \in \mathit{rul}_{P(A,B)}}$$

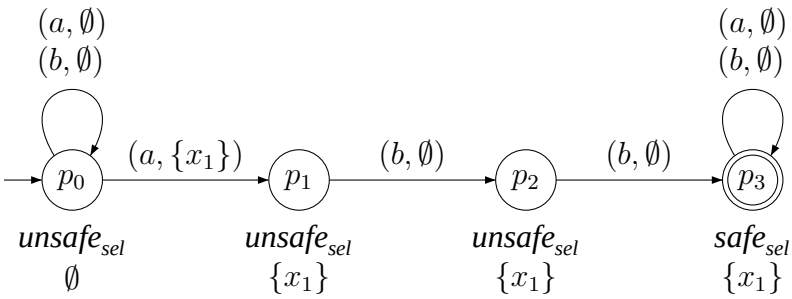


Figure 7.2: Automaton A for the query selecting a-nodes followed by $b{\cdot}b$. There are two reachable unsafe states of type $\{x_1\} = \mathcal{V}_1$, p_1 and p_2. The restriction of A to these two states is acyclic, so the selection delay of Q_A is bounded. It is bounded by 2, since the longest path in this part of the automaton has 2 nodes.

Building $P(A, B)$ needs time in $O((|\Sigma|+n)\cdot|A|\cdot|B|)$, if we suppose for instance that variables in v are stored in a vector of n bits.

Lemma 17. *Let A and B be productive dFAs that define a query, and r a run of $P(A, B)$ on $w * \tau$ and $\eta \in eve(w)$. Then state $r(\eta)$ is safe for selection by $Q_{A,B}$ if and only if $(\tau, \eta) \in sel_{Q_{A,B}}(w)$.*

Proof. Sufficiency for selection $(\tau, \eta) \in sel_{Q_{A,B}}(w)$ is equivalent to $\tau \in dom_\eta(w)^n$ and $\forall w' \in L(B) :\ equal_\eta(w, w') \Rightarrow w' * \tau \in L(A)$. Let $w = w_0{\cdot}w_1$ such that $|w_0| = \eta$. Since $\tau \in dom_\eta(w)^n$, we have $w * \tau = (w_0 * \tau){\cdot}(w_1 \otimes \emptyset)$. Furthermore, $equal_\eta(w, w')$ is equivalent to $\exists w_1'.\ w' = w_0{\cdot}w_1'$. Now $r(\eta)$ is the state that the unique run of $P(A, B)$ on $w_0 * \tau$ reaches (determinism). For $(p, q) = r(\eta)$ we have:

$$
\begin{aligned}
& \forall w' \in L(B) :\ equal_\eta(w, w') \Rightarrow w' * \tau \in L(A) \\
\Leftrightarrow\ & \forall w_1'.\ w_0{\cdot}w_1' \in L(B) \Rightarrow (w_0 * \tau){\cdot}(w_1' \otimes \emptyset) \in L(A) \\
\Leftrightarrow\ & \forall w_1'.\ w_1' \in L(B[init = \{q\}]) \Rightarrow w_1' \otimes \emptyset \in L(A[init = \{p\}]) \quad \text{(determinism)} \\
\Leftrightarrow\ & L(B[init = \{q\}]) \otimes \emptyset \subseteq L(A[init = \{p\}]) \\
\Leftrightarrow\ & r(\eta) \text{ safe for selection by } Q_{A,B}
\end{aligned}
$$

Conversely, assume that $r(\eta) = (p, q)$ is safe for selection by $Q_{A,B}$. Since we assumed A and B to be productive, this implies that $type(p) = \mathcal{V}_n$, so that $\tau \in dom_\eta(w)^n$. We can thus decompose $w = w_0 \cdot w_1$ such that $|w_0| = \eta$ as above, and apply the above equivalence, in order to conclude from safety for selection, that $\forall w' \in L(B) :\ equal_\eta(w, w') \Rightarrow w' * \tau \in L(A)$, and thus sufficiency for selection. □

The parallel automaton $P(A, B)$ is canonical, since $L(A) = L(P(A, B))$, but may contain non-productive states, even if A and B are productive. For instance,

consider productive automata A and B that define the query Q with $\mathit{dom}(Q) = \{a, a{\cdot}a\}$, $Q(a) = \{1\}$ and $Q(a{\cdot}a) = \emptyset$. We will be interested only in the productive part of the canonical automaton $P(A, B)$, for which unique types exist.

Lemma 18. *If A and B are productive, then all safe states of $Q_{A,B}$ that are reachable in $P(A, B)$ are productive and have type $\mathcal{V}_n$.*

Proof. To see this, suppose that (p, q) is safe and reachable. Since B is productive, there exists a word $w \in L(B[\mathit{init}{=}\{q\}])$. Safety proves that $w \otimes \emptyset \in L(A[\mathit{init} = \{p\}])$. Thus, $w \in P(A, B)[\mathit{init} = \{(p, q)\}]$, so that (p, q) is productive. Since A is canonical, $P(A, B)$ is canonical, so that $\mathit{type}(p) \uplus \mathit{type}(w \otimes \emptyset) = \mathcal{V}_n$. □

Capturing the Delay

Proposition 24. *Let $Q_{A,B}$ be defined by productive dFAs A and B, and let P^u be the restriction of nFA $P(A, B)$ to productive unsafe states of type $\mathcal{V}_n$.*

1. *The delay of $Q_{A,B}$ is bounded if and only if the digraph of nFA P^u is acyclic.*
2. *In this case, the delay of $Q_{A,B}$ is equal to the length of the longest path in P^u.*

Proof. Let $P = P(A, B)$ and P^u the restriction of P to productive unsafe states of type $\mathcal{V}_n$. Let q be a state of P^u for which a cycle exists. Since all states of P^u are productive in P, there exists a word $v_1 \in L(P[\mathit{fin} = \{q\}])$. Since P^u has a cycle, there exists a nonempty word $v_2 \in L(P[\mathit{init} = \{q\}, \mathit{fin} = \{q\}])$. Again, since P is productive, there exists a word $v_3 \in L(P[\mathit{init} = \{q\}])$. It follows for all $m \geq 0$, that $v = v_1{\cdot}(v_2)^m{\cdot}v_3 \in L(P)$. Since $L(P) = L(A)$, word v has the form $w * \tau$ for some word $w \in \Sigma^*$ and $\tau \in \mathit{dom}(w)^n$. By Lemma 17, none of the events in $|v_2|^m$ is sufficient for the selection of τ in w since the run of P on v maps all of them to unsafe states. This shows that the selection delay of τ in v is at least m and thus unbounded.

For the converse, we suppose that P^u is acyclic and show that the delay of $Q_{A,B}$ is bounded by the length of the longest path in stat_{P^u}. Let w and τ be such that $w * \tau \in L(A)$ and r be the successful run of A that accepts this word. Let η be an arbitrary event that contributes to the delay of τ, i.e., an event with $\tau \in \mathit{dom}_\eta(w)$ and $(\tau, \eta) \notin \mathit{sel}_{Q_{A,B}}(w)$. The first condition yields that $\mathit{type}(r(\eta)) = 1^{\mathcal{V}_n}$ and the second condition that $r(\eta)$ is unsafe for selection by Lemma 17. Thus, $r(\eta) \in \mathit{stat}_{P^u}$. Since P^u is acyclic, it follows that states $r(\eta)$ are distinct for distinct events η that contribute to the delay. Furthermore, all these states belong to the same path of P^u, such that $\mathit{delay}_{Q_{A,B}}(w, \tau)$ is bounded by the length of the longest path in P^u.

If P^u is acyclic, let r a longest path in P^u and let w a word such that $w * \emptyset$ labels r. Since all states of P are reachable and productive, there exists $w_1 * \tau$ which reaches in P the first state of r; similarly, there exists a word w_2 such that $w_2 * \emptyset$ labels a path from the last state of r to a final state of P. Then $\mathit{delay}_Q(w_1 \cdot w \cdot w_2, \tau)$ is the length (here, the number of states) of r.

□

In order to compute the set of all safe states, we rely on the following characterization of unsafe states.

Lemma 19. *Let A, B be productive dFAs that define a query. A reachable state (p_0, q_0) of $P(A, B)$ is unsafe for selection by $Q_{A,B}$ if and only a state (p, q) can be reached from (p_0, q_0) such that:*

*(*U1*) either $p \notin \mathit{fin}_A$ and $q \in \mathit{fin}_B$,*

*(*U2*) or there exists a transition $q \xrightarrow{a} q' \in \mathit{rul}_B$ but no transition $p \xrightarrow{(a,\emptyset)} p' \in \mathit{rul}_A$ for all $p' \in \mathit{stat}_A$.*

Proof. Let $P = P(A, B)$. We start with a claim about propagation of unsafety.

Claim 14. *Reachable states of P that can reach unsafe states are unsafe.*

To see this, let (p_1, q_1) be a reachable state and (p_2, q_2) be an unsafe state that is reached from (p_1, q_1) by some word v_1, i.e. $v_1 \in P[\mathit{init} = \{(p_1, q_1)\}, \mathit{fin} = \{(p_2, q_2)\}]$. Since (p_2, q_2) is unsafe, there exists a word $w \in L(B[\mathit{init} = \{q_2\}])$ such that $w \otimes \emptyset \notin L(A[\mathit{init} = \{p_2\}])$. We distinguish two cases.

1. If v_1 matches $w_1 \otimes \emptyset$ then $w_1 \cdot w \in L(B[\mathit{init} = \{q_1\}])$ and $(w_1 \cdot w) \otimes \emptyset \notin L(A[\mathit{init} = \{p_2\}])$, so that (p_1, q_1) is unsafe.

2. If v_1 does not match $w_1 \otimes \emptyset$ then $\mathit{type}(p_1) \neq \mathcal{V}_n$ so that (p_1, q_1) is unsafe by Lemma 18, since (p_1, q_1) is reachable in P and since A and B are productive.

Based on this claim, we can now show both directions of the lemma.

"⇐" By Claim 14 it is sufficient to show that all states (p, q) satisfying (U1) or (U2) are unsafe. In case of (U1) where $p \notin \mathit{fin}_A$ and $q \in \mathit{fin}_B$, the empty word contradicts the safety of (p, q), since $\epsilon \in L(B[\mathit{init} = \{q\}])$ but $\epsilon \otimes \emptyset \notin L(A[\mathit{init} = \{p\}])$. In case of (U2), there exists some transition $q \xrightarrow{a} q' \in \mathit{rul}_B$ but no transition $p \xrightarrow{(a,\emptyset)} p' \in \mathit{rul}_A$ for all $p' \in \mathit{stat}_A$. Since B is productive, there exists a word $w \in L(B[\mathit{init} = \{q_2\}])$. The word $a \cdot w$ now contradicts safety of (p, q) since $a \cdot w \in L(B[\mathit{init} = \{p\}])$ but $(a \cdot w) \otimes \emptyset \notin L(A[\mathit{init} = \{q\}])$.

"⇒" We show that all unsafe states (p_0, q_0) can reach some state (p, q) that satisfies (U1) or (U2). If (p_0, q_0) is unsafe then there exists a word $w \in \Sigma^*$ such that $w \in L(B[\mathit{init} = \{q_0\}])$ and $w \otimes \emptyset \notin L(A[\mathit{init} = \{p_0\}])$. Let w_0 be the longest prefix of w such that there exists a run of $P[\mathit{init} = \{(p_0, q_0)\}]$ on w_0. Let (p, q) be the state reached by this run after reading w_0, and let w_1 be the suffix of w such that $w = w_0 \cdot w_1$. State (p, q) is thus reached from (p_0, q_0). It remains to show that (p, q) satisfies (U1) or (U2).

1. If $w_1 = \epsilon$ then $p \in \mathit{fin}_B$ and $q \notin \mathit{fin}_A$, so that (p, q) satisfies (U1).
2. If w_1 matches $a \cdot w_2$ then there cannot exist any transition $p \xrightarrow{(a,\emptyset)} p'$ since w_0 was chosen of maximal length. There exists a transition $q \xrightarrow{a} q'$ for some q' though. Hence, (p, q) satisfies (U2).

□

Lemma 20. *The set of reachable safe states for selection for an n-ary query $Q_{A,B}$ can be computed in time $O((|\Sigma| + n) \cdot |A| \cdot |B|)$ from dFAs A and B.*

Proof. Instead of the set of reachable safe states, we compute the set of reachable unsafe states. A Datalog program testing the reachability of states satisfying (U1) or (U2), which characterizes unsafety for reachable states by Lemma 19, can be defined as follows:

$$\frac{p' \notin \mathit{fin}_A \qquad q' \in \mathit{fin}_B}{\mathit{unsafe}_{\mathit{sel}}(p, q).} \qquad \frac{\forall p'.\, p \xrightarrow{(a,\emptyset)} p' \notin \mathit{rul}_A \qquad q \xrightarrow{a} q' \in \mathit{rul}_B}{\mathit{unsafe}_{\mathit{sel}}(p, q).}$$

$$\frac{(p, q) \xrightarrow{(a,V)} (p', q') \in \mathit{rul}_{P(A,B)}}{\mathit{unsafe}_{\mathit{sel}}(p, q) \text{ :- } \mathit{unsafe}_{\mathit{sel}}(p', q').}$$

This program P can be computed in time $O((|\Sigma| + n) \cdot |A| \cdot |B|)$, while being of size $O(|A| \cdot |B|)$. It is a ground Datalog program, so its least fixed point $\mathit{lfp}(P)$ can be computed in time $O(|A| \cdot |B|)$ (see Proposition 5 in the appendix). □

Theorem 10. *The delay of queries $Q_{A,B}$ in words with alphabet Σ and arity $n \in \mathbb{N}_0$ defined by dFAs A and B can be computed in time $O((|\Sigma| + n) \cdot |A| \cdot |B|)$.*

In particular, we can decide in the same time, whether a query $Q_{A,B}$ has bounded delay or k-bounded delay, even if k belongs to the input.

Proof. We first render B productive and construct the dFA $P(A, B)$. Second, we compute all reachable safe states by Lemma 20 and derive the sub-automaton P^u, that restricts $P(A, B)$ to productive unsafe states of type $\mathcal{V}_n$. By Proposition 24, the delay of $Q_{A,B}$ is ∞ if and only if P^u contains a cycle. Otherwise, we compute the delay by counting the length of the longest path of P^u. All of these operations can be performed in time $O((|\Sigma| + n) \cdot |A| \cdot |B|)$. □

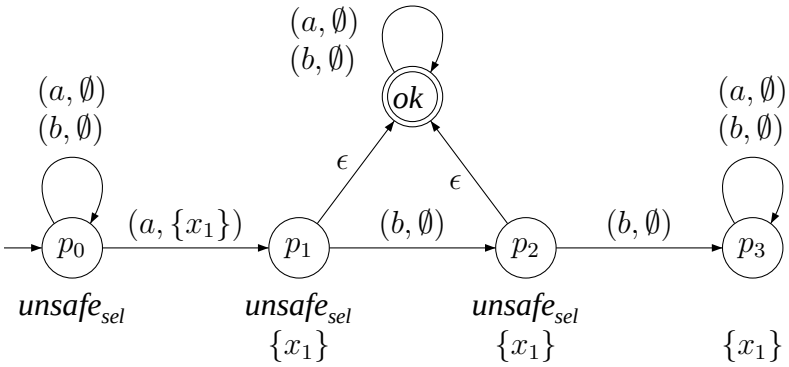


Figure 7.3: nFA $D(A,B)$ for the dFA A in Figure 7.2 with trivial universal B. The ambiguity of $D(A,B)$ is 2 (on word $(a,\{x_1\})\cdot(b,\emptyset)$ for instance), such as the delay of $Q_{A,B}$.

7.3.4 Reduction to Bounded Ambiguity

Before moving on to bounded concurrency, we introduce a more general method to decide boundedness by reduction to bounded ambiguity of nFAs at the example of bounded delay.

The idea is to turn the dFA $P(A,B)$ it into an nFA $D(A,B)$ such that $amb_{D(A,B)}(w * \tau) = delay_{Q_{A,B}}(w,\tau)$ for all $\tau \in Q_{A,B}(w)$. We can then test whether $D(A,B)$ has bounded or k-bounded ambiguity, which can be done in PTIME as shown in [AMR08, Sei92].

We construct $D(A,B)$ from $P(A,B)$ by adding a new state *ok* and ϵ-transitions from all unsafe states of type $\mathcal{V}_n$ to *ok*. Figure 7.3 presents the result of this operation on the automaton in Figure 7.2.

$$stat_{D(A,B)} = stat_{P(A,B)} \uplus \{ok\},\ init_{D(A,B)} = init_{P(A,B)},\ fin_{D(A,B)} = \{ok\}$$

$$\frac{r \in rul_{P(A,B)}}{r \in rul_{D(A,B)}} \qquad \frac{unsafe_{sel}(p,q) \qquad p \text{ has type } \mathcal{V}_n}{(p,q) \xrightarrow{\epsilon} ok \in rul_{D(A,B)}} \qquad \frac{a \in \Sigma}{ok \xrightarrow{(a,\emptyset)} ok \in rul_{D(A,B)}}$$

Proposition 25. *For all $\tau \in Q_{A,B}(w)$: $delay_{Q_{A,B}}(w,\tau) = amb_{D(A,B)}(w * \tau)$.*

Proof. Consider a run r of $D(A,B)$ on a canonical word $w * \tau$ with $\tau \in Q(w)$. We can show inductively on r that the ambiguity of $D(A,B)$ on w is exactly the number of states used in r that are not safe for selection. The initial state is unique as A is deterministic, so at the beginning the ambiguity is 1. When reading a new letter, if the associated state q is not unsafe or has not type $\mathcal{V}_n$, then there is only one way to continue the run, via a rule of $P(A,B)$. If it is unsafe with type $\mathcal{V}_n$,

then there are two possibilities: either by using the run of $P(A, B)$, or by firing the ϵ-transition. Both runs will succeed (as *ok* is universal), so in this case the ambiguity is increased by one. Hence $amb_{D(A,B)}(w * \tau)$ is the number of unsafe states used in the run of $P(A, B)$, and also of A, on $w * \tau$. From the definitions of delay (here the type $\mathcal{V}_n$ ensures that we start counting at $latest(\tau)$), safe states and by Lemma 17, this is exactly $delay_{Q_{A,B}}(w, \tau)$. □

Proposition 25 yields slightly weaker results than Theorem 10. It permits to apply PTIME algorithms for deciding bounded or k-bounded ambiguity of dFAs, in order to decide bounded or k-bounded delay in PTIME. However, it does *not* allow to compute the optimal bound in P-time, requires to fix k in order to decide k-boundedness in P-time, and yields higher polynomials. As we will show next, this general approach is useful to decide bounded and k-bounded concurrency, for which we do not dispose any more direct algorithm.

7.3.5 Deciding Bounded Concurrency

We show how to reduce in PTIME bounded concurrency to bounded ambiguity and k-bounded concurrency to k-bounded ambiguity.

The concurrency of a query counts the number of simultaneously alive partial candidates. Beside of sufficiency for selection, aliveness depends on sufficiency for rejection. We thus need a notion of safe states for rejection.

Definition 15. *A pair of states (p, q) of $P(A, B)$ is* safe for rejection by $Q_{A,B}$ *if no final state can be reached from (p, q), i.e., if $L(P(A, B)[init = \{(p, q)\}]) = \emptyset$.*

We saw in the proof of Theorem 10 how to compute safe states for selection, so now we need a method to compute safe states for rejection.

Lemma 21. *The set of safe states for rejection by $Q_{A,B}$ for nFAs A and B can be computed in time $O(|A| \cdot |B|)$.*

Proof. We compute the set of all unsafe states for rejection. In order to do so, it is sufficient to compute the set of all states of $P(A, B)$ from which some final state can be reached. This can be done by the following ground Datalog program:

$$\frac{p' \in fin_A \qquad q' \in fin_B}{unsafe_{rej}(p, q).} \qquad \frac{p \xrightarrow{(a,v)} p' \in rul_A \qquad q \xrightarrow{a} q' \in rul_B}{unsafe_{rej}(p, q) \text{ :- } unsafe_{rej}(p', q').}$$

This program can be constructed in time $O(|A| \cdot |B|)$ from A and B. By Proposition 5, the $lfp(P)$ can be computed in time $O(|A| \cdot |B|)$. □

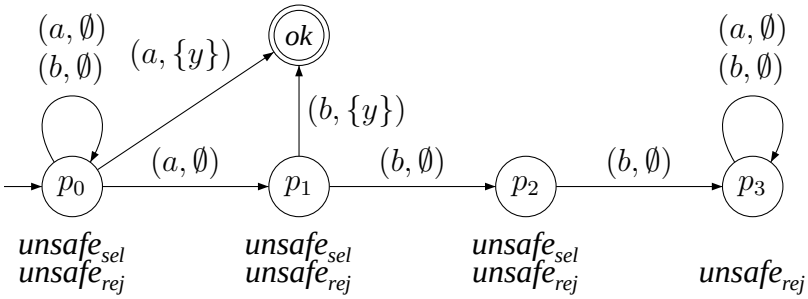


Figure 7.4: nFA $C(A,B)$ for query dFA A in Figure 7.2 and trivial universal B. Even though nondeterministic, the ambiguity of $C(A,B)$ is 1, equally to the concurrency of $Q_{A,B}$.

We define an nFA $C(A,B)$ such that $amb_{C(A,B)}(w * \eta) = concur_Q(w,\eta)$. The situation is a little different than for $D(A,B)$, in that $C(A,B)$ runs on words annotated by events rather than tuples. We fix a new variable $y \notin \mathcal{V}_n$ that will denote the event of interest, and define the alphabet of $C(A,B)$ to be $\Sigma \times 2^{\{y\}}$. The idea of nFA $C(A,B)$ is to guess a partial candidate τ, until the event marker y comes, and to test whether τ is alive at that event, and to accept in case of success.

$$stat_{C(A,B)} = stat_A \times stat_B \uplus \{ok\}$$
$$init_{C(A,B)} = init_A \times init_B$$
$$fin_{C(A,B)} = \{ok\}$$

$$\frac{(p,q) \overset{(a,V)}{\to} (p',q') \in rul_{P(A,B)}}{(p,q) \overset{(a,\emptyset)}{\to} (p',q') \in rul_{C(A,B)}}$$

$$\frac{(p,q) \overset{(a,v)}{\to} (p_1,q_1) \in rul_{P(A,B)} \quad unsafe_{sel}(p_1,q_1) \quad unsafe_{rej}(p_1,q_1)}{(p,q) \overset{(a,\{y\})}{\to} ok \in rul_{C(A,B)}}$$

Both rules guess a set of variables V and check that the current position is the denotation of all variables in V, by running automaton A with V in the input letter. The second rules inputs the event marker, and goes into the *ok*-state, if automaton $P(A,B)$ could move to states that are unsafe for both selection and rejection, so that the current partial candidate is alive. For illustration, consider Figure 7.4 which shows the automaton $C(A,B)$ obtained from the automaton A in Figure 7.2 and the trivial universal automaton B.

Given a word $w = a_1 \cdot \ldots \cdot a_m$ and a position $1 \leq \eta \leq m$ we write $w|\eta$ for the word $(a_1,\emptyset) \cdot (a_{\eta-1},\emptyset) \cdot (a_\eta,\{y\})$.

Proposition 26. *$concur_{Q_{A,B}}(w,\eta) = amb_{C(A,B)}(w|\eta)$, for all $w \in L(B)$ and $\eta \in dom(w)$.*

Proof. Let $w \in L(B)$ and $\eta \in dom(w)$. Suppose that τ_1 and τ_2 are different partial tuples that are alive at η. Let r_1 and r_2 be the runs of A on the prefixes of

$w * \tau_1$ resp. $w * \tau_2$ until η. Since τ_1 and τ_2 are different, there exists a position i such that the prefixes of length $i < \eta$ of $w * \tau_1$ and $w * \tau_2$ have different types. Since A is canonical, this implies that both runs assign states of different types to position i, so that $r_1(i) \neq r_2(i)$.

Let $a_1 \cdot \ldots \cdot a_\eta$ be the prefix of w until position η. By construction of $C(A, B)$, both runs r_i restricted to $\{1, \ldots, \eta-1\}$ are also runs of $C(A, B)$ on word $v = (a_1 \cdot \ldots \cdot a_{\eta-1}) \otimes \emptyset$. These runs can be extended to successful runs of $C(A, B)$ on $w|\eta = v \cdot (a_\eta, \{y\})$ by mapping position η to *ok*, since both tuples τ_i are alive at event η (and thus neither safe for selection nor rejection). Both runs are different, since runs r_1 and r_2 differ at some position $i < \eta$. Hence $\mathit{concur}_{Q_{A,B}}(w, \eta) \leq \mathit{amb}_{C(A,B)}(w|\eta)$.

For the converse, consider two different runs r_1 and r_2 of $C(A, B)$ on $w|\eta$. We now build two partial tuples τ_1 and τ_2 and the corresponding runs r'_1 and r'_2 of A on the prefixes of $w * \tau_1$ and $w * \tau_2$ until η. These are hidden in the rules applied for producing runs r_1 and r_2 by $C(A, B)$. Since the states which permitted to move to *ok* are alive, the runs r'_1 and r'_2 can be extended into an alive state at η. This shows that both tuples τ_1 and τ_2 are alive. They are different, since produced from distinct runs r_1 and r_2. This shows that $\mathit{amb}_{C(A,B)}(w|\eta) \leq \mathit{concur}_{Q_{A,B}}(w, \eta)$. □

Theorem 11. *Bounded and k-bounded concurrency for queries and schemas defined by canonical dFAs can be decided in* PTIME *for any fixed $k \geq 0$.*

Proof. From Lemmas 20 and 21, $C(A, B)$ can be constructed in PTIME from A and B. By Proposition 26, it remains to decide the finite (resp. k-bounded) ambiguity of $C(A, B)$. This can also be done in PTIME [AMR08, Sei92]. Before the construction, we need to make A and B productive, which can be done in time $O(|A| + |B|)$. □

7.4 Recognizable Relations between Unranked Trees

Even with STAs, it remains difficult to lift our PTIME algorithms for words to trees, since the notion of safe states becomes more complex. The difference is that in STAs, the configuration depends on the current state, but also on the content of the stack. Given a canonical dSTA A for query Q_A, one can define another dSTA $E(A)$ for which appropriate notions of safe states w.r.t. Q_A exist, as shown in Chapter 5. The size of $E(A)$, however, may grow exponentially in $|A|$. Therefore, we cannot use $E(A)$ to construct polynomially sized counterparts of $D(A)$ and $C(A)$ in the case of unranked trees, for instance automata which ambiguity captures the delay (resp. the concurrency). We conjecture that in the general case

there is no PTIME algorithm for computing deterministic automata capturing the delay and the concurrency from A.

Nevertheless, we are able to prove the following theorem:

Theorem 12 (Main)**.** *Bounded delay is decidable in* PTIME *for n-ary queries defined by deterministic streaming tree automata where n may be variable. Bounded concurrency is decidable in* PTIME *for fixed n. For fixed k and n, k-bounded delay and concurrency are decidable in* PTIME.

Since top-down deterministic tree automata (d↓TAs) modulo *fcns* encoding and bottom-up deterministic automata (dTAs) modulo *curry* encoding can be translated to dSTAs in PTIME (see Chapter 4), Theorem 12 does equally apply for queries defined by such automata. The proof will be based on reductions to bounded resp. k-bounded valuedness of recognizable relations between unranked trees. It will be presented in Section 7.5.

Regular tree languages enjoy closure properties over logical operations, thanks to the underlying properties of tree automata. A tree language can be considered as a unary relation over the set of all trees. A generalization consists in considering n-ary relations over trees, i.e., sets of n-tuples of trees.

In this section, we show how to extend the notion of recognizable relations [CDG$^+$07] to the case of unranked trees. Closure properties of automata still ensure that FO-formulas over recognizable relations with n free variables define recognizable relations between n unranked trees (so that satisfiability is decidable). Unlike the framework proposed by Benedikt et al. [BLN07], we do not define basic relations, and allow different alphabets on the components of the relations. Our major contribution here is that bounded valuedness and k-bounded valuedness (for a fixed k) of binary relations can be decided in PTIME. For bounded valuedness, we use a reduction to bounded valuedness of transducers [Sei92]. k-bounded valuedness is resolved by reduction to the emptiness of an automaton, that can be computed in PTIME thanks to properties of recognizable relations.

7.4.1 Closure Properties

Cylindrification Extension

Cylindrification of queries has been defined in Section 2.3, as the inverse projection. We extend the definition in order to allow the insertion of several components (instead of one), plus copying and permutation of components, but no deletion. For an n-ary query Q over relational structures $\mathcal{S}$, cylindrification $c_\theta Q$ for a function $\theta : \{1, \ldots, m\} \to \{1, \ldots, m\}$ with $\{1, \ldots, n\} \subseteq \theta(\{1, \ldots, m\})$ is defined by the following equality, for all structures $s \in \mathcal{S}$:

$$c_\theta Q(s) = \{(\pi_{\theta(1)}, \ldots, \pi_{\theta(m)}) \in \mathit{dom}(s)^m \mid (\pi_1, \ldots, \pi_n) \in Q(s)\}$$

The schema is unchanged: $\mathit{dom}(c_\theta Q) = \mathit{dom}(Q)$.

Queries by First-Order Formulas

In Section 2.3 and in the previous paragraph, we defined logical operations on queries. We show how they can be used to define queries from first-order formulas. This is an alternative definition of first-order definable queries introduced in Section 2.3.

Every FO formula ϕ with at most m free variables $\tilde{y} = (y_1, \ldots, y_m) \in \mathcal{V}^m$ defines a m-ary query $Q_{\phi(\tilde{y})}$ whose domain contains all $\mathcal{S}$-structures.

$$Q_{\phi_1 \wedge \phi_2(\tilde{y})} = Q_{\phi_1(\tilde{y})} \wedge Q_{\phi_2(\tilde{y})} \qquad Q_{\neg\phi(\tilde{y})} = \neg Q_{\phi(\tilde{y})}$$
$$Q_{(\exists z.\phi)(\tilde{y})} = \exists z.Q_{\phi(\tilde{y},z)} \qquad Q_{r(y_1,\ldots,y_n)(y_{\theta(1)},\ldots,y_{\theta(m)})} = c_\theta r$$

Here, we identify relation symbol r with the query of arity $\mathit{ar}(r)$ that satisfies $r(s) = r^s$ for all structures $s \in \mathcal{S}$.

Logical Operations on Tree Languages

Beyond standard Boolean operations on languages [CDG+07], we define projection operations proj_i: $\mathcal{T}_{\Sigma_1 \times \ldots \times \Sigma_m} \to \mathcal{T}_{\Sigma_i}$ for all $1 \leq i \leq m$, such that all $\mathit{proj}_i(t)$ relabels all nodes $\pi \in \mathit{nod}(t)$ to the i-th component of its label. We write $t = t_1 * \cdots * t_m$ if $\wedge_{1 \leq i \leq m} \mathit{proj}_i(t) = t_i$. We can define more general projection operations proj_I: $\mathcal{T}_{\Sigma_1 \times \ldots \times \Sigma_m} \to \mathcal{T}_{\Sigma_{i_1} \times \ldots \times \Sigma_{i_n}}$ that preserve a subset of components $I = \{i_1, \ldots, i_n\}$ where $1 \leq i_1 < \ldots < i_n \leq m$ by $\mathit{proj}_I(t_1 * \ldots * t_m) = t_{i_1} * \ldots * t_{i_n}$. Projections can be lifted to languages of trees $L \subseteq \mathcal{T}_{\Sigma_1 \times \ldots \times \Sigma_m}$ by $\mathit{proj}_I(L) = \{\mathit{proj}_I(t) \mid t \in L\}$.

We also need cylindrification operations on tree languages, which may add, copy, and exchange components of tuple trees, but not delete them. We formalize *unsorted* cylindrification operations that apply to trees $L \subseteq \mathcal{T}_{\Sigma^n}$, where all components have the same signature Σ. For functions $\theta : \{1, \ldots, m\} \to \{1, \ldots, m\}$ with $\{1, \ldots, n\} \subseteq \theta(\{1, \ldots, m\})$ we define:

$$c_\theta L = \{t_{\theta(1)} * \ldots * t_{\theta(m)} \in \mathcal{T}_{\Sigma^m} \mid t_1 * \ldots * t_n \in L\}$$

Note that all newly added components have signature Σ. *Sorted* cylindrification operations, that add components of particular types, can be obtained from unsorted cylindrification and intersection.

Closure Properties of Automata

In this chapter, we assume an arbitrary class of tree automata, that satisfy the properties in Proposition 27. In particular, we consider three classes of tree automata studied in Chapter 4: TAs w.r.t. *fcns* and *curry* encodings, and STAs. They

all have the same expressiveness, as proved by the back and forth translations in Chapter 4, and exactly capture MSO-definable queries (and languages) over unranked trees. In the following, we say that a tree language is *recognizable* if it is MSO-definable.

Proposition 27 (Closure properties)**.** *Recognizable languages are closed under Boolean operations, projection and cylindrification. All corresponding operations on tree automata can be performed in* PTIME*, except for the complementation of non-deterministic tree automata. They all preserve determinism except for projection.*

Proof. Closure properties of recognizable languages are due to the closure properties of MSO-definable languages. It is folklore that these operations are in PTIME and preserve determinism except for projection, for the three classes of automata we consider.

Cylindrification operations c_θ are a little richer than the usual cylindrification operations c_i that insert a single new component at position i [CDG$^+$07]. In addition, they can copy components, which can be tested by intersection with deterministic tree automata that recognize the set $\{t * t \mid t \in \mathcal{T}_\Sigma\}$, and permute components. While operation c_θ can be implemented in PTIME for every fixed θ by computing intersections with a fixed number of tree automata, this cannot be done in PTIME for variable θ.

Note, however, that cylindrification cannot delete components, such as projection, since projection operations on automata may spoil determinism. □

7.4.2 Recognizable Relations

We study recognizable relations between trees [CDG$^+$07] in the ranked and unranked case [BLN07]. These are sets of tuples of trees, such that the set of overlays of these tuples is recognizable by a tree automaton.

We first recall a standard method to define recognizable relations in FO logic from a set of basic recognizable relations, while relying on the closure properties of tree automata. We then present the second main contribution of this article. We show that bounded valuedness and k-bounded valuedness (for a fixed k) of binary relations can be decided in PTIME. For bounded valuedness, we present a PTIME reduction to bounded valuedness of transducers [Sei92], and for k-bounded valuedness, a PTIME reduction to emptiness of tree automaton.

In this section, we assume an arbitrary class of automata for unranked trees $\mathcal{A}$ that satisfy the following properties. Here, we assume that every automaton $A \in \mathcal{A}$ has an abstract notion of states s_A.

(A1) every automaton of $\mathcal{A}$ can be transformed into an STA in PTIME.

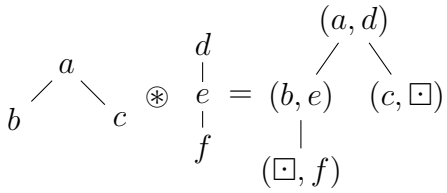


Figure 7.5: Example for overlays

(A2) class $\mathcal{A}$ is closed under intersection, complementation, cylindrification and projection modulo PTIME transformations, that preserve determinism except for projection.

All these properties hold for the three classes of automata studied in the previous section: Chapter 4 proves the expressiveness requirement (A1) and Proposition 27 the closure properties (A2). Note however, that hedge automata with dFAs for horizontal languages [CDG+07] fail to satisfy (A2), since deterministic hedge automata cannot be complemented in PTIME.

The *overlay* of k unranked trees $t_i \in \mathcal{T}_{\Sigma^i}$ is the unranked tree $t_1 \circledast \ldots \circledast t_k$ in $\mathcal{T}_{\Sigma^1_\boxdot \times \ldots \times \Sigma^k_\boxdot}$ obtained by superposing these k trees top-down and left-to-right; the $\boxdot$ symbol represents missing children where the structures of the trees differ. This is illustrated in Figure 7.5 and formally defined by:

$$a(t_1, \ldots, t_k) \circledast b(t'_1, \ldots, t'_l) = \begin{cases} (a,b)(t_1 \circledast t'_1,\ \ldots,\ t_l \circledast t'_l,\ t_{l+1} \circledast \boxdot,\ \ldots,\ t_k \circledast \boxdot) & \text{if } l \leq k \\ (a,b)(t_1 \circledast t'_1,\ \ldots,\ t_k \circledast t'_k,\ \boxdot \circledast t_{k+1},\ \ldots,\ \boxdot \circledast t_l) & \text{otherwise} \end{cases}$$

Overlays of ranked trees can be obtained this way too [CDG+07], except that overlayed symbols need to inherit the maximal arity.

Definition 16. *A k-ary relation R between unranked trees is* recognizable *iff the language of its overlays $ovl(R) = \{t_1 \circledast \ldots \circledast t_k \mid (t_1, \ldots, t_k) \in R\}$ is recognizable by a tree automaton. We say that R is recognized by the automaton A if $ovl(R) = L(A)$.*

Prime examples for recognizable relations [BLN07] are the tree extension relation $\leq_\downarrow, \leq_\rightarrow \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$, such that $t \leq_\downarrow t'$ if t' is obtained by repeatedly adding children to leaves of t, and $t \leq_\rightarrow t'$ if t' is obtained by repeatedly adding next-siblings to right most children of t.

7.4.3 Sorted FO Logic

We need a sorted first-order logic in order to define recognizable relations between trees with various signatures. Note that only the simpler case with a single signature was treated in [BLN07].

A *sorted relational signature* is a relational signature $\mathcal{S} = \mathit{Sorts} \uplus \Re$, that consists of a set of monadic symbols $\sigma \in \mathit{Sorts}$ called sorts and a set of relation symbols $r \in \Re$, each of which has a sort $\mathit{sort}(r) \in \mathit{Sorts}^{ar(r)}$. A *sorted relational structure* s over $\mathcal{S} = \mathit{Sorts} \uplus \Re$ is a relational structure such that: $\mathit{dom}(s) = \cup_{\sigma \in \mathit{Sorts}} \sigma^s$ and for every relation symbol $r \in \Re$ of arity m:

$$\mathit{sort}(r) = (\sigma_1, \ldots, \sigma_m) \Rightarrow r^s \subseteq \sigma_1^s \times \ldots \times \sigma_m^s$$

In the FO logic of sorted relational structures, we can define sort bounded quantifiers:

$$\exists x{\in}\sigma.\phi =_{df} \exists x.(\sigma(x) \wedge \phi)$$

A *sorted FO formula* is a FO formula in which all quantifiers are sort bounded. Every sorted FO formula ϕ over $\mathcal{S}$ with at most m free sorted variables defines an n-ary relation for every sorted relational structure s over $\mathcal{S}$:

$$R_{\phi(x_1:\sigma_1,\ldots,x_m:\sigma_m)}(s) = Q_{\phi(x_1,\ldots,x_m)}(s) \cap \sigma_1^s \times \ldots \times \sigma_m^s$$

7.4.4 Sorted FO Logic of Recognizable Relations

We assume a collection of alphabets Ω. A structure s of recognizable relations between trees with alphabets in Ω has a sorted relational signature with sorts $\mathit{Sorts} = \{\mathcal{T}_\omega \mid \omega \in \Omega\}$ that are interpreted by themselves in every structure, such that every relation symbol $r \in \Re$ is interpreted as a recognizable relation $r^s \subseteq \mathit{sort}(r)$ between trees.

A sorted FO formula for recognizable relations with alphabets Ω has the following form where $r \in \Re$ and $T_1, \ldots, T_n, T \in \mathcal{V}$ and $\omega \in \Omega$.

$$\phi \ ::= \ r(T_1, \ldots, T_{ar(r)}) \mid \phi \wedge \phi' \mid \neg\phi \mid \exists T{\in}\mathcal{T}_\omega.\ \phi$$

Here we use capital letters for variables, since they range over trees rather than nodes of a single tree. The size $|\phi|$ of a formula is the number of nodes of ϕ.

We write $\mathrm{FO}_\exists[\Re]$ for the set of sorted formulas, where quantifiers are existential and in prenex positions. Let $s = \{A_r\}_{r \in \Re}$ be a collection of automata that recognize the relations in $\Re$, or equivalently, the structure of recognizable relations they induce. Every sorted FO formula ϕ with at most m free sorted variables defines an n-ary relation between trees:

$$R_{\phi(T_1:\mathcal{T}_{\omega_1},\ldots,T_m:\mathcal{T}_{\omega_m})}(\vartheta) \subseteq \mathcal{T}_{\omega_1} \times \cdots \times \mathcal{T}_{\omega_m}$$

The closure properties of tree automata w.r.t. Boolean operations, cylindrification, and projection ensure that all such relations are recognizable.

Proposition 28. *Let ϕ be a fixed formula in FO$_\exists[\Re]$ with at most m free sorted variables $T_1{:}\mathcal{T}_{\omega_1}, \ldots, T_m{:}\mathcal{T}_{\omega_m}$. Then there exists a polynomial p such that for all structures of recognizable relations $\vartheta = \{A_r\}_{r\in\Re}$ defined by tree automata such that A_r is deterministic if r occurs below in negation ϕ, one can compute in time $p(\sum_{r\in\Re}|A_r|)$ an automaton that recognizes the relation $R_{\phi(T_1:\mathcal{T}_{\omega_1},\ldots,T_m:\mathcal{T}_{\omega_m})}(\vartheta)$. The computed automaton is deterministic, if all automata are deterministic and ϕ is free of existential quantifiers.*

Proof. The proposition depends of the closure properties (A2) of the class of automata under consideration. The proof is by induction on the structure of formulas in FO$_\exists[\Re]$. It follows from two claims, that relate operations on tree relations to operations on tree languages to closure properties of tree automata.

Claim 17. *For all $Q \subseteq \mathcal{T}_{\omega_1} \times \ldots \times \mathcal{T}_{\omega_m}$, $\mathcal{V}_m = \{X_1, \ldots, X_m\}$ and $\theta : \{1,\ldots,m\} \to \{1,\ldots,m\}$ with $\{1,\ldots,n\} \subseteq \theta(\{1,\ldots,m\})$:*

$$\begin{array}{ll} \mathit{ovl}(\exists X_i.Q) = \mathit{proj}_{\{1,\ldots,i-1,i+1,\ldots,m\}}(\mathit{ovl}(Q)) & \mathit{ovl}(c_\theta Q) = c_\theta \mathit{ovl}(Q) \\ \mathit{ovl}(\neg Q) = \mathit{ovl}(\mathcal{T}_{\omega_1} \times \ldots \times \mathcal{T}_{\omega_m}) - \mathit{ovl}(Q) & \mathit{ovl}(Q_1 \wedge Q_2) = \mathit{ovl}(Q_1) \cap \mathit{ovl}(Q_2) \end{array}$$

The proof is straightforward from the definitions. The next second claim relates connectives of sorted FO formulas to operations on tree relations.

Claim 18. *For all alphabets $\tilde{\omega} = (\omega_1, \ldots, \omega_m)$ and ω_{m+1}, variables $\tilde{X} = (X_1, \ldots, X_m)$ and X_{m+1} that are pairwise distinct, structures s of tree relations, functions $\theta : \{1,\ldots,m\} \to \{1,\ldots,m\}$ with $\{1,\ldots,n\} \subseteq \theta(\{1,\ldots,m\})$, sorted formulas ϕ, ϕ_1, ϕ_2 in FO$[\Re]$, and relations symbols $r \in \Re$:*

$$\begin{array}{l} \mathit{ovl}(R_{\exists X_{m+1}\in\mathcal{T}_{\omega_{m+1}}.\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) = \mathit{proj}_{\{1,\ldots,m\}}(\mathit{ovl}(R_{\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}},X_{m+1}:\mathcal{T}_{\omega_{m+1}})}(s))) \\ \mathit{ovl}(R_{r(\tilde{X})(X_{\theta(1)}:\mathcal{T}_{\omega_{\theta(1)}},\ldots,X_{\theta(m)}:\mathcal{T}_{\omega_{\theta(m)}})}(s)) = \mathit{ovl}(\mathcal{T}_{\omega_{\theta(1)}} \times \ldots \times \mathcal{T}_{\omega_{\theta(m)}}) \cap c_\theta \mathit{ovl}(r^s) \\ \mathit{ovl}(R_{\phi_1\wedge\phi_2(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) = \mathit{ovl}(R_{\phi_1(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) \cap \mathit{ovl}(R_{\phi_2(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) \\ \mathit{ovl}(R_{\neg\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) = \mathit{ovl}(\mathcal{T}_{\omega_1} \times \ldots \times \mathcal{T}_{\omega_m}) - \mathit{ovl}(R_{\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) \end{array}$$

The proof is straightforward from the definitions and the previous claim. For illustration, we elaborate the case of negation, where the sorting information is needed. Let $L_{\tilde{\omega}} = \mathit{ovl}(\mathcal{T}_{\omega_1} \times \ldots \times \mathcal{T}_{\omega_m})$.

$$\begin{array}{rcll} \mathit{ovl}(R_{\neg\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) & = & L_{\tilde{\omega}} \cap \mathit{ovl}(Q_{\neg\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) & \\ & = & L_{\tilde{\omega}} \cap (L_{\tilde{\omega}} - \mathit{ovl}(Q_{\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s))) & \text{(previous claim)} \\ & = & L_{\tilde{\omega}} - \mathit{ovl}(R_{\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s)) & \end{array}$$

Finally, we illustrate the induction for formula $\phi = \neg\phi'$. Since $\phi \in$ FO$_\exists[\Re]$, formula ϕ' cannot contain existential quantifiers. Furthermore, all automata A_r for relations symbols occurring in ϕ must be deterministic by assumption. By induction

hypothesis, there exists a polynomial p' such that for all structures $\vartheta = \{A_r\}_{r\in\Re}$ defined automata automata A_r, one can compute in time $p(\sum_{r\in\Re}|A_r|)$ a deterministic automaton A' recognizing the language $\mathit{ovl}(R_{\phi'(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(\vartheta))$. Recall that $\mathit{ovl}(R_{\phi(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(\vartheta))$ is equal to $\mathit{ovl}(\mathcal{T}_{\omega_1}\times\ldots\times\mathcal{T}_{\omega_m}) - \mathit{ovl}(R_{\phi'(\tilde{X}:\mathcal{T}_{\tilde{\omega}})}(s))$ as shown by the previous claim. We obtain an automaton A recognizing this language by complementing A' and intersecting it with an automaton for $\mathit{ovl}(\mathcal{T}_{\omega_1}\times\ldots\times\mathcal{T}_{\omega_m})$. This can be done in time $p_1(|A'|)\cdot|\omega_1|\cdot\ldots\cdot|\omega_m|$ for some polynomial p_1, since A' was deterministic. Furthermore, automaton A can be constructed deterministically from A'. We can thus define polynomial p by $p(\xi) = p_1(p'(\xi))\cdot|\omega_1|\cdot\ldots\cdot|\omega_m|$.

The only construction, where non-determinism is needed are projections. This is why we require existential quantifiers to appear only in prenex position. Note that the proposition can be extended to general FO formulas, but not in PTIME. □

In Section 7.5, we will see that relations capturing the notions of delay and concurrency of queries $Q_{A,B}$ can be defined in PTIME from A and B by using $\mathrm{FO}_\exists[\Re]$ formulas, for a suitable set of relation symbols $\Re$ whose interpretation depends on A and B. The delay and concurrency will exactly be the valuedness of the corresponding recognizable relations. In the remainder of this section, we prove that bounded valuedness and k-bounded valuedness or recognizable relations are decidable in PTIME from automata defining the relations.

7.4.5 Bounded Valuedness

Let $R \subseteq \mathcal{T}_{\Sigma_1}\times\mathcal{T}_{\Sigma_2}$ be a recognizable binary relation. For every $t_1 \in \mathcal{T}_{\Sigma_1}$, the number $\#R(t_1) = |\{t_2 \mid (t_1,t_2)\in R\}|$ counts the trees in $\mathcal{T}_{\Sigma_2}$ in relation to it. The *valuedness* of R is the maximal such number $\mathit{val}(R) = \max_{t\in\mathcal{T}_{\Sigma_1}}\#R(t)$. We call R *k-bounded* if $\mathit{val}(R)\leq k$, and *bounded* if it is k-bounded for some $k\in\mathbb{N}_0$.

We want to reduce bounded valuedness of recognizable relations over unranked trees to the same problem for ranked trees. This can be obtained by a correspondence between the overlay of a tree and the overlay of its *fcns* encoding. Let *ren* be the morphism on binary trees that renames constants $(\boxdot,\ldots,\boxdot)$ to $\boxdot$ and preserves the trees otherwise. This morphism is linear and one-to-one, so it preserves regularity in both directions: L is recognizable iff $\mathit{ren}(L)$ is recognizable. The following lemma relates overlays of unranked and ranked trees. Note that this nice correspondence does not hold for the *curry* encoding.

Lemma 22. $\mathit{fcns}(t_1\circledast\ldots\circledast t_n) = \mathit{ren}(\mathit{fcns}(t_1)\circledast\ldots\circledast\mathit{fcns}(t_n))$

The following proposition shows that valuedness is preserved by the *fcns* encoding. Let $\mathit{fcns}(R) = \{(\mathit{fcns}(t_1),\mathit{fcns}(t_2)) \mid (t_1,t_2)\in R\}$.

Proposition 29. *A binary relation R between unranked trees is recognizable iff the corresponding relation between binary trees $\mathit{fcns}(R)$ is, and $\mathit{val}(\mathit{fcns}(R)) = \mathit{val}(R)$.*

Proof. By definition $\mathit{fcns}(R) = \{(\mathit{fcns}(t_1), \mathit{fcns}(t_2)) \mid (t_1, t_2) \in R\}$. Lemma 22 yields $\mathit{fcns}(\mathit{ovl}(R)) = \mathit{ren}(\mathit{ovl}(\mathit{fcns}(R)))$. The morphism *ren* preserves recognizability back and forth. Thus, $\mathit{fcns}(R)$ is a recognizable relation iff $\mathit{ovl}(\mathit{fcns}(R))$ is recognizable language of binary trees iff $\mathit{ren}(\mathit{ovl}(\mathit{fcns}(R)))$ is a recognizable language of binary trees iff $\mathit{fcns}(\mathit{ovl}(R))$ is a recognizable language of binary trees iff $\mathit{ovl}(R)$ is a recognizable language of unranked trees iff R is a recognizable relation of unranked trees. □

Theorem 13. *For every automaton A recognizing a binary relation R between unranked trees, $\mathit{val}(R) < \infty$ can be decided in* PTIME *in $|A|$.*

This theorem holds for all classes of automata for unranked trees that satisfy the expressiveness property (A1) and thus to kinds of tree automata introduced before. Note that we will apply this theorem to non-deterministic automata A later on.

Proof. We prove Theorem 13 in two steps. First we show by Proposition 30 that the result holds for relabeling relations. A relabeling relation $R \subseteq \mathcal{T}_{\Sigma^1} \times \ldots \times \mathcal{T}_{\Sigma^n}$ is a relation between trees of the same structure, i.e. whenever $(t_1, \ldots, t_n) \in R$ then $\mathit{nod}(t_1) = \ldots = \mathit{nod}(t_n)$. In other words, the overlays in $\mathit{ovl}(R)$ do not contain any place holder $\boxdot$. Then we exhibit how to associate with any relation R a relabeling relation C_R with the same valuedness, where the automaton recognizing C_R can be constructed in PTIME from A defining R. The correctness of the construction is proved by Lemma 24. □

Proposition 30. *The finite valuedness of a binary relabeling recognizable relation R can be decided in* PTIME *in $|A|$, when given an automaton A recognizing R.*

Proof. Every automaton can be converted to an STA in PTIME by assumption (A1), and thus to a TAs modulo the *fcns* encoding by translations of Chapter 4. Proposition 29 permits to reduce the current Proposition to recognizable relations of binary trees defined by standard TAs.

So let $R \subseteq \mathcal{T}^{\mathit{bin}}_{\Sigma_1} \times \mathcal{T}^{\mathit{bin}}_{\Sigma_2}$ be a relabeling relation for binary signatures, and A a TA for trees in $\mathcal{T}^{\mathit{bin}}_{\Sigma_1 \times \Sigma_2}$ that recognizes R, i.e. $L(A) = \mathit{ovl}(R)$. We transform A into a bottom-up tree transducer T for defining the relation R of the format in [Sei92]. The rules of T are inferred as follows where x_1, x_2 are variables:

$$\frac{(f,g)(q_1,q_2) \to q \in \mathit{rul}_A}{f(q_1(x_1), q_2(x_2)) \to q(g(x_1,x_2)) \in \mathit{rul}_T} \qquad \frac{(a,b) \to q \in \mathit{rul}_A}{a \to q(b) \in \mathit{rul}_T}$$

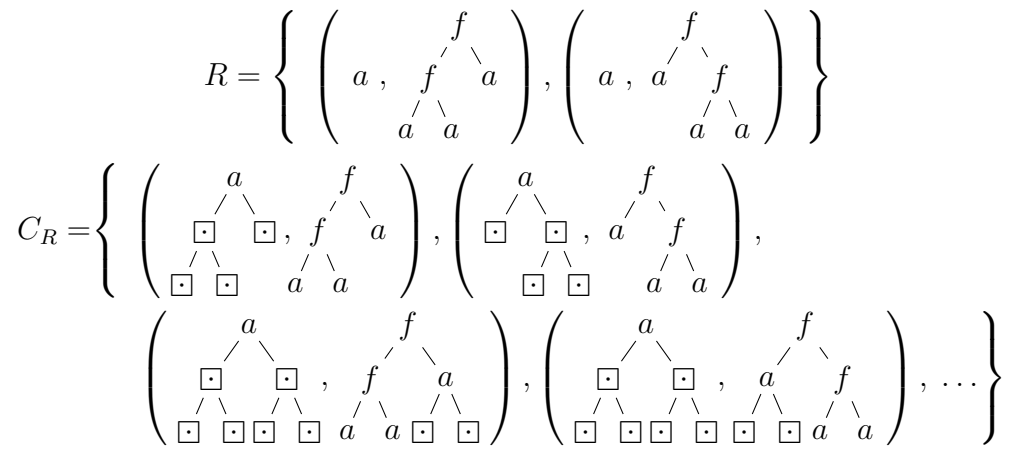


Figure 7.6: A recognizable relation R and the relabeling C_R with the same valuedness.

This transducer T has the same valuedness as R. Theorem 2.8 of [Sei92] shows that it can be decided in polynomial time whether T is finite-valued, i.e. whether R is bounded. □

The above construction of bottom-up transducers cannot be lifted to recognizable relations beyond relabelings. Instead, we show how to convert recognizable relations into recognizable relabelings, while preserving valuedness.

So, let R be a recognizable relation over $\mathcal{T}^{bin}_{\Sigma^1} \times \mathcal{T}^{bin}_{\Sigma^2}$. We define a recognizable relabeling $C_R \in \mathcal{T}^{bin}_{\Sigma^1_\boxdot \times \Sigma^2_\boxdot}$, where we have 2 symbols $(\boxdot, \boxdot)$ with arities 0 and 2 respectively. The idea is to expand both trees in pairs $(t_1, t_2) \in R$ to trees $(t'_1, t'_2) \in C_R$ of the same structure, by repeatedly adding $\boxdot$-children to leaves of t_1 or t_2. Expansion $ex_i(t, t')$ holds for two trees $t \in \mathcal{T}^{bin}_{\Sigma^i}$ and $t' \in \mathcal{T}^{bin}_{\Sigma^i_\boxdot}$ if $nod(t) \subseteq nod(t')$, both trees have the same labels on common nodes, and all new nodes of t' are labeled by $\boxdot$. We define the relabeling C_R by:

$$C_R = \{(t'_1, t'_2) \in \mathcal{T}_{\Sigma^1_\boxdot} \times \mathcal{T}_{\Sigma^2_\boxdot} \mid (t_1, t_2) \in R,\ ex_1(t_1, t'_1),\ ex_2(t_2, t'_2),\ nod(t'_1) = nod(t'_2)\}$$

An example is given in Figure 7.6. While the relation R there is finite, the corresponding relabeling C_R is infinite, since it has infinitely many witnesses of every pair of R.

Lemma 23. *If A is a dTA recognizing R, then there exists a dTA A' of size $O(|A|)$ that recognizes C_R.*

Proof. We add one more state to A, so that $stat(A') = stat(A) \cup \{q_\boxdot\}$ and $fin(A) = fin(A')$. Automaton A' runs A top-down, until $\boxdot$ occurs, and then checks for equal

domains:

$$(\boxdot,\boxdot) \to q_{\boxdot} \in \mathit{rul}(A') \qquad (\boxdot,\boxdot)(q_{\boxdot},q_{\boxdot}) \to q_{\boxdot} \in \mathit{rul}(A')$$

$$\frac{(a,b) \to q \in \mathit{rul}_A}{(a,b)(q_{\boxdot},q_{\boxdot}) \to q \in \mathit{rul}(A') \quad (a,b)(q_{\boxdot}) \to q \in \mathit{rul}(A')}$$

□

Lemma 24. *C_R and R have the same valuedness.*

Proof. If $\mathit{ex}_i(t,t')$ holds for $(t,t') \in \mathcal{T}_{\Sigma^i} \times \mathcal{T}_{\Sigma^i_{\boxdot}}$, then we write $\mathit{clean}_i(t') = t$, which is well-defined as t is unique for a given t'. It is easy to check that:

- if $s \in \mathcal{T}_{\Sigma^1_{\boxdot} \times \Sigma^2_{\boxdot}}$ then $s \in \mathit{ovl}(C_R)$ iff $(\mathit{clean}_1(\mathit{proj}_1(s)), \mathit{clean}_2(\mathit{proj}_2(s))) \in R$
- $(t_1,t_2) \in C_R$ iff $(\mathit{clean}_1(t_1), \mathit{clean}_2(t_2)) \in R$ and $\mathit{nod}(t_1) = \mathit{nod}(t_2)$.

First, let us prove that the valuedness of C_R is at least the valuedness of R. Let t in $\mathcal{T}_{\Sigma^1}$ such that there exists at least k distinct t_i with $(t,t_i) \in R$. Let $D = \mathit{nod}(t) \cup \cup_{i=1}^{k} \mathit{nod}(t_i)$. For a tree u and a set of nodes D such that $\mathit{nod}(t) \subseteq D$, we define the completion of u w.r.t. D as the tree u^D defined by $\mathit{nod}(u^D) = D$ and $\mathit{lab}^{u^D}(\pi) = \mathit{lab}^u(\pi)$ if p belongs to $\mathit{nod}(u)$, $\mathit{lab}^{u^D}(\pi) = \boxdot$ otherwise. As $\mathit{nod}(t^D) = \mathit{nod}(t_i^D)$ and $\mathit{clean}_1(t^D) = t, \mathit{clean}_2(t_i^D) = t_i$, we have $(t^D, t_i^D) \in C_R$, $1 \leq i \leq n$. As the t_i, $1 \leq i \leq n$, are distinct, so are the t_i^D, $1 \leq i \leq n$: the valuedness of C_R is at least the valuedness of R.

Now, let us prove that the valuedness of C_R is at most the valuedness of R. Let u in $\mathcal{T}_{\Sigma^1_{\boxdot}}$ such that there exists at least k distinct v_i with $C_R(u,v_i)$. Let $t = \mathit{clean}_1(u), t_i = \mathit{clean}_2(v_i)$: we have $(t,t_i) \in R$. It remains to prove that the t_i are all distinct.

Let $1 \leq i < j \leq n$: as $v_i \neq v_j$ there exists a position π such that $\mathit{lab}^{v_i}(\pi) \neq \mathit{lab}^{v_j}(\pi)$:

- either $\mathit{lab}^{v_i}(\pi) \neq \boxdot$ and $\mathit{lab}^{v_j}(\pi) \neq \boxdot$: then π belongs to $\mathit{nod}(t_i)$ and to $\mathit{nod}(t_j)$ and $\mathit{lab}^{t_i}(\pi) \neq \mathit{lab}^{t_j}(\pi)$.
- either $\mathit{lab}^{v_i}(\pi) \neq \boxdot$ and $\mathit{lab}^{v_j}(\pi) = \boxdot$: then π belongs to $\mathit{nod}(t_i)$ and π does not belong to $\mathit{nod}(t_j)$.
- either $\mathit{lab}^{v_j}(\pi) \neq \boxdot$ and $\mathit{lab}^{v_i}(\pi) = \boxdot$: similar to the precedent case.

So, there exists $t \in \mathcal{T}_{\Sigma^1}$ such that there exists at least k distinct t_i with $(t,t_i) \in R$. □

Even if testing bounded valuedness of tree transducers is known to be in PTIME, the complexity of known polynomial algorithms is much higher than for testing bounded ambiguity of tree automata [SdS08].

Note that if we add the condition that A is deterministic, then a similar construction could have been done using automata instead of transducers. If A' is the automaton on Σ_2 obtained from A by projecting the Σ_1 components, then $\mathit{amb}(A') = \mathit{val}(R)$, and ambiguity and k-ambiguity of A' can be obtained in PTIME [Sei92]. However, we will use relations defined by $\mathrm{FO}_\exists[\Re]$ formula, which corresponding automata are non-deterministic.

7.4.6 k-Bounded Valuedness

In this section we study the problem of deciding whether a binary recognizable relation has k-bounded valuedness. We first prove that, when k is fixed, we can still decide k-bounded valuedness in PTIME. Then we consider the problem when k is variable, and prove that it becomes EXPTIME-hard.

Here we cannot prove that k-bounded valuedness can be decided in PTIME through the use of transducers, like for Lemma 30, as known algorithms for deciding k-boundedness of transducers are in non-deterministic polynomial time (Theorem 2.2 of [Sei92]).

The problem does neither reduce to deciding the k-ambiguity of an automaton. We will need to measure the valuedness of relations (as they will capture delay and concurrency), but $\mathit{amb}(A)$ and $\mathit{val}(R)$ are not comparable, when A recognizes R.

Theorem 14. *Let Σ_1 and Σ_2 be two alphabets and $k \in \mathbb{N}_0$ fixed. There exists a polynomial p such that for every structure $\mathfrak{s}$ with a single relation $R \subseteq \mathcal{T}_{\Sigma_1} \times \mathcal{T}_{\Sigma_2}$ recognized by a possibly nondeterministic tree automaton A, $\mathit{val}(R) \leq k$ can be decided in time $p(|A|)$.*

Proof. We consider the tree relation $\mathit{SameTree} = \{(t,t) \mid t \in \mathcal{T}_{\Sigma_2}\}$ which is recognizable by a tree automaton of size $O(|\Sigma_2|^2)$. We fix a binary relation symbol r that is interpreted by structures $\mathfrak{s}$ given by R such that $r^{\mathfrak{s}} = R$. We define a formula $\mathit{val}_{>k}$ with $k+2$ free variables in the logic of recognizable relations in $\mathrm{FO}_\exists[r, \mathit{SameTree}]$, such that $R_{\mathit{val}_{>k}(T:\mathcal{T}_{\Sigma_1}, T_1:\mathcal{T}_{\Sigma_2}, \ldots T_{k+1}:\mathcal{T}_{\Sigma_2})}(R) = \emptyset$ if and only if $\mathit{val}(R) > k$:

$$\mathit{val}_{>k} =_{\mathit{df}} \bigwedge_{1 \leq i \leq k+1} r(T, T_i) \wedge \bigwedge_{1 \leq i < j \leq k+1} \neg \mathit{SameTree}(T_i, T_j)$$

A tree automaton recognizing relation $R_{\mathit{val}_{>k}(T:\mathcal{T}_{\Sigma_1}, T_1:\mathcal{T}_{\Sigma_2}, \ldots T_{k+1}:\mathcal{T}_{\Sigma_2})}(R) = \emptyset$ can be computed in polynomial time from tree automaton A, where the polynomial

depends on the fixed parameters $|\Sigma_1|$, $|\Sigma_2|$ and k. This follows from Proposition 28 since formula relation symbol r does not occur below negation in formula $val_{>k}$. Emptiness of the language of this automaton can be tested in linear time. Hence, there exists a polynomial p (depending on the fixed parameters k, Σ_1, and Σ_2), such that we can check $val(R) > k$ in polynomial time $O(p(|A|))$ from an automaton A recognizing R. □

Theorem 14 provides a PTIME decision procedure k-bounded valuedness, under the assumption that k is fixed and the proof relies on an automaton of size $O(|A|^{k+1})$. Without this assumption, however, we cannot avoid an exponential blow-up.

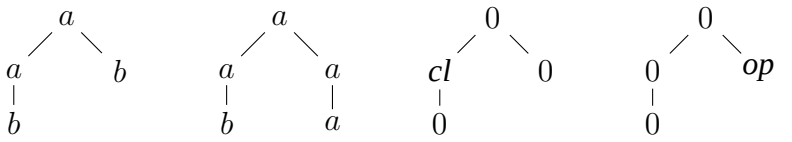
Theorem 15. *The problem that inputs $k \in \mathbb{N}_0$ and an automaton A recognizing a binary relation R between unranked trees, and outputs the truth value of $val(R) \leq k$ is* EXPTIME*-complete.*

Proof. By the proof of Theorem 14, the problem is in EXPTIME. For the hardness part, we will reduce emptiness of intersection of deterministic tree automata in this problem. Let $Int(S)$ the problem that inputs S, a finite sequence of deterministic tree automata, and outputs "yes" if and only if there is at least one term recognized by each automaton of the sequence. Now, from all automata A we can build in polynomial time a binary relation R_A that associates with a tree t, t labeled by an accepting run, if such a run exists. So, from S - w.l.o.g. we suppose the set of states are disjoint- we construct in polynomial time an automaton A_S for the binary relation $\vee_{A \in S} R_A$. As the automata are deterministic, A_S will be $(|S| - 1) - bounded$ iff there isn't any term recognized by each automaton of the sequence. We conclude as emptiness of intersection of deterministic tree automata is EXPTIME-hard. □

Using the above constructions and Theorem 2.7 of [Sei92], we can build an algorithm for computing the exact value of $val(R)$, if it exists. The overall complexity is a fixed number of exponentials in $|A|$.

7.5 Deciding Bounded Delay and Concurrency

We prove the main Theorem 12 on deciding bounded delay and concurrency for queries defined by dSTAs by reduction to bounded valuedness of recognizable relations.



(a) Tree t (b) Tree s (c) Tree $\mathit{ren}^{\eta}(t)$ (d) Tree $\mathit{ren}^{\eta'}(t)$

Figure 7.7: $(t, s, \mathit{ren}^{\eta}(t)) \in \mathit{Eq}$ but $(t, s, \mathit{ren}^{\eta'}(t)) \notin \mathit{Eq}$

7.5.1 Basic Recognizable Relations

We start by defining various relations between trees by dSTAs, that we will use later on for defining the delay and concurrency of dSTA defined queries by recognizable relations between trees.

The prime example is the tree relation $\mathit{Eq} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \times \mathcal{T}_{\{0,op,cl\}}$. For every event $\eta = (\alpha, \pi) \in \mathit{eve}(t)$ and tree $t \in \mathcal{T}_\Sigma$, let $\mathit{ren}^{\eta}(t) \in \mathcal{T}_{\{0,op,cl\}}$ be obtained by renaming the label of π to α and the labels of all other nodes of t to 0. We then define:

$$(t, s, \mathit{ren}^{\eta}(t)) \in \mathit{Eq} \Leftrightarrow_{df} \mathit{equal}_{\eta}(t, s)$$

so that t and s have the same prefix until event η. See Figure 7.7 for an example.

Lemma 25. *For every signature Σ we can compute a dSTA in time $O(|\Sigma|^2)$, that recognizes the relation $\mathit{Eq} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \times \mathcal{T}_{\{0,op,cl\}}$.*

Proof. We define a dSTA A on $\Sigma_\boxdot \times \Sigma_\boxdot \times \{0, \mathit{op}, \mathit{cl}\}_\boxdot$ such that $L(A) = \mathit{ovl}(\mathit{Eq})$. We use two states $\mathit{stat}_{\mathrm{e}}^A = \{\mathit{before}, \mathit{after}\}$, where $\mathit{init}^A = \{\mathit{before}\}$ and $\mathit{fin}^A = \{\mathit{after}\}$. We use a single dummy node state $\mathit{stat}_{\mathrm{n}}^A = \{\_\}$. The rules are given by the following inference schema:

$$\frac{\alpha \in \{\mathit{op}, \mathit{cl}\} \qquad a \in \Sigma \qquad b \in \Sigma_\boxdot}{\begin{array}{ll} \mathit{before} \xrightarrow{\alpha\ (a,a,0):\_} \mathit{before} & \mathit{before} \xrightarrow{\mathit{op}\ (a,a,\mathit{cl}):\_} \mathit{before} \\ \mathit{before} \xrightarrow{\alpha\ (a,a,\alpha):\_} \mathit{after} & \mathit{after} \xrightarrow{\alpha\ (a,b,0):\_} \mathit{after} \\ \mathit{after} \xrightarrow{\mathit{cl}\ (a,b,\mathit{op}):\_} \mathit{after} & \mathit{after} \xrightarrow{\alpha\ (\boxdot,a,\boxdot):\_} \mathit{after} \end{array}}$$

Note that the rule $\mathit{before} \xrightarrow{\mathit{op}\ (a,a,\mathit{cl}):\_} \mathit{before}$ is used to check the equality below a node π if prefix equality has to be checked until (cl, π). Automaton A has size $O(|\Sigma^2|)$ and can be computed in this time. □

The next kind of tree relations express canonical languages of queries. Given a tree $t \in \mathcal{T}_\Sigma$ and a complete tuple $\tau \in \mathit{dom}(t)^n$, we define a tree $\mathit{prune}^{\tau}(t) \in \mathcal{T}_{2^{\mathcal{V}_n}}$ as follows. Let t' be the prefix of t with domain $\mathit{dom}_{\mathit{latest}(\tau)}(t)$. We set $\mathit{prune}^{\tau}(t) = \mathit{proj}_2(t' * \tau)$.

For every n-ary query Q, we define a recognizable relation $\mathit{Can}_Q \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_{2^{\mathcal{V}_n}}$, which relates trees $t \in \mathcal{T}_\Sigma$ with tuples $\tau \in Q(t)$:

$$\mathit{Can}_Q = \{(t, \mathit{prune}^\tau(t)) \mid \tau \in Q(t)\}$$

Lemma 26. *Let A and B be dSTAs that define an n-ary query $Q = Q_{A,B}$. Then we can compute a dSTA from A in time $O(|A|^2 \cdot |\Sigma|)$ that recognizes Can_Q.*

Note that the size of the computed automaton is independent of n, even though $2^{\mathcal{V}_n}$ appears in the alphabet of Can_Q.

Proof. An automaton A_C recognizing Can_Q can be built in polynomial time in $|A|$ and $|\Sigma|$. The idea is exploit the types of states of canonical automata, in order to detect event $\eta = \mathit{latest}(\tau)$, rather than storing the variables seen so far in the state. In order to ensure the uniqueness of types, we have to make A productive. We can then compute the types of all states during a traversal of the automaton. The automaton A_C can then be computed as follows:

$$\begin{array}{l} \mathit{stat}_{\mathrm{e}}^{A_C} = \mathit{stat}_{\mathrm{e}}^{A} \\ \mathit{init}^{A_C} = \mathit{init}^{A} \\ \mathit{fin}^{A_C} = \mathit{fin}^{A} \\ \mathit{stat}_{\mathrm{n}}^{A_C} = \mathit{stat}_{\mathrm{n}}^{A} \end{array} \qquad \frac{q_0 \xrightarrow{\alpha\ (a,v):\gamma} q_1 \in \mathit{rul}_A \qquad q_0 \text{ not of type } \mathcal{V}_n \text{ or } \alpha = \mathit{cl}}{q_0 \xrightarrow{\alpha\ (a,v):\gamma} q_1 \in \mathit{rul}_{A_C}}$$

$$\frac{q_0 \xrightarrow{\alpha\ (a,\emptyset):\gamma} q_1 \in \mathit{rul}_A \qquad q_0 \text{ of type } \mathcal{V}_n}{q_0 \xrightarrow{\alpha\ (a,\boxdot):\gamma} q_1 \in \mathit{rul}_{A_C}}$$

The automaton simulates A until it reaches states of type $\mathcal{V}_n$. From there on, it expects $\boxdot$ as annotation, instead of $\emptyset$. Note that A_C is deterministic since A is. □

The relation $\mathit{Bef} = \{(t, \mathit{prune}^\tau(t), \mathit{ren}^\eta(t)) \mid \tau \in \mathit{dom}_\eta(t)^n\}$ is the subset of $\mathcal{T}_\Sigma \times \mathcal{T}_{2^{\mathcal{V}_n}} \times \mathcal{T}_{\{0,\mathit{op},\mathit{cl}\}}$ that captures all n-tuples of nodes of t (on its second component) that contain only nodes opened before an event η provided by the third component. *Bef* is recognizable by a dTA of size $O(2^n)$, so we cannot use this relation for PTIME algorithms without fixing n. The problem can be circumvented by using the following relation $\mathit{Bef\&Can}_Q$ which can be recognized while using the states of the canonical automaton for $L(Q)$ for checking types:

$$\mathit{Bef\&Can}_Q = \{(t, s_\tau, s_\eta) \in \mathcal{T}_\Sigma \times \mathcal{T}_{2^{\mathcal{V}_n}} \times \mathcal{T}_{\{0,\mathit{op},\mathit{cl}\}} \mid \mathit{Can}_Q(t, s_\tau),\ \mathit{Bef}(t, s_\tau, s_\eta)\}$$

Lemma 27. *We can compute a dSTA $A_{B\&C}$ recognizing $\mathit{Bef\&Can}_{Q_{A,B}}$ in time $O(|A|^2 \cdot |\Sigma|)$.*

Proof. We build a dSTA $A_{B\&C}$ that recognizes *Bef*&*Can*$_Q$ in PTIME from the dSTA A_C recognizing *Can*$_Q$. We have to check, that at most one event η is annotated into the third component, and that is comes after *latest*(τ) for the tuple τ of the second component, i.e., when automaton A_C has moved into a state of type $\mathcal{V}_n$.

Let $\mathbb{B} = \{0, 1\}$ be the set of Booleans. We define $\mathit{stat}_{\mathrm{e}}^{A_{B\&C}} = \mathit{stat}_{\mathrm{e}}^{A_C} \times \mathbb{B}$, in order to control by a Boolean, whether the third component has been seen before. We define initial states by $\mathit{init}^{A_{B\&C}} = \mathit{init}^{A_C} \times \{0\}$, final states by $\mathit{fin}^{A_{B\&C}} = \mathit{fin}^{A_C} \times \{1\}$, and node states by $\mathit{stat}_{\mathrm{n}}^{A_{B\&C}} = \mathit{stat}_{\mathrm{n}}^{A_C}$.

$$\frac{q_0 \xrightarrow{\alpha\ (a,v):\gamma} q_1 \in \mathit{rul}_{A_C} \qquad \flat \in \mathbb{B} \qquad \alpha' \neq \alpha}{(q_0, \flat) \xrightarrow{\alpha\ (a,v,\alpha'):\gamma} (q_1, \flat) \in \mathit{rul}_{A_{B\&C}}}$$

$$\frac{q_0 \xrightarrow{\alpha\ (a,v):\gamma} q_1 \in \mathit{rul}_{A_C} \qquad q_1 \text{ has type } \mathcal{V}_n \text{ in } A_C}{(q_0, 0) \xrightarrow{\alpha\ (a,v,\alpha):\gamma} (q_1, 1) \in \mathit{rul}_{A_{B\&C}}}$$

□

We define a variant of *Bef* for partial tuples, called *Bef*$_\bullet$. Here, we do not try to avoid the blow-up for two reasons. First, *Bef*$_\bullet$ will be used with another relation called $C_{2^{\mathcal{V}_n}}$, and a blow-up is necessary to recognize $C_{2^{\mathcal{V}_n}}$. Second, separating the relations permits to clarify the definition of the formula capturing concurrency. Let $\mathit{ren}^\tau(s) \in \mathcal{T}_{2^{\mathcal{V}_n}}$ be the projection of $s * \tau$ to $2^{\mathcal{V}_n}$, i.e., $\mathit{nod}(\mathit{ren}^\tau(s)) = \mathit{nod}(s)$ and $\mathit{lab}^{\mathit{ren}^\tau(s)}(\pi) = v$ if $\mathit{lab}^s(\pi) = (a, v)$ for some $a \in \Sigma$, and all $\pi \in \mathit{nod}(s)$.

The relation $\mathit{Bef}_\bullet = \{(\mathit{ren}^\tau(t), \mathit{ren}^\eta(t)) \mid \exists t \in \mathcal{T}_\Sigma.\ \tau \in \mathit{dom}_\eta^\bullet(t)^n\}$ is a subset of $\mathcal{T}_{2^{\mathcal{V}_n}} \times \mathcal{T}_{\{0,op,cl\}}$ that relates annotations of trees with tuples τ and events η, such that $\mathit{latest}(\tau) \preceq \eta$.

Lemma 28. *A dSTA recognizing* *Bef*$_\bullet$ *can be computed in time* $O(3^n)$.

Proof. The following dSTA $A_{\mathit{Bef}_\bullet}$ recognizes the relation *Bef*$_\bullet$. In the states, we collect (at opening) variables corresponding to the components of τ that have been encountered. We also add a Boolean, that indicates whether the event η has been read. Note that on the second component, we can read values different from 0 when we are not at η. For instance if $\eta = (\mathit{op}, \pi)$, we will read "*op*" on the second component when we go through (cl, π).

$$\mathit{stat}_{\mathrm{e}}^{A_{\mathit{Bef}_\bullet}} = 2^{\mathcal{V}_n} \times \mathbb{B} \qquad \mathit{init}^{A_{\mathit{Bef}_\bullet}} = \{(\emptyset, 0)\} \qquad \mathit{fin}^{A_{\mathit{Bef}_\bullet}} = 2^{\mathcal{V}_n} \times \{1\} \qquad \mathit{stat}_{\mathrm{n}}^{A_{\mathit{Bef}_\bullet}} = \{\_\}$$

Rules are defined by the following inference schemas. At opening, we check canonicity if η has not been reached; otherwise we forbid variables in the first

component. When η is reached, we still allow to read variables, and change the Boolean.

$$\frac{\alpha \in \{0, \mathit{cl}\} \qquad v, v' \subseteq \mathcal{V}_n \qquad v \cap v' = \emptyset}{\begin{array}{c}(v,0) \xrightarrow{\mathit{op}\ (v',\alpha):\_} (v \cup v', 0) \in \mathit{rul}^{A_{\mathit{Bef}\bullet}} \\ (v,0) \xrightarrow{\mathit{op}\ (v',\mathit{op}):\_} (v \cup v', 1) \in \mathit{rul}^{A_{\mathit{Bef}\bullet}} \\ (v,1) \xrightarrow{\mathit{op}\ (\emptyset,0):\_} (v, 1) \in \mathit{rul}^{A_{\mathit{Bef}\bullet}}\end{array}}$$

At closing, we do not check anything. We just change the Boolean when η is reached.

$$\frac{\flat \in \mathbb{B} \qquad \alpha \in \{0, \mathit{op}\} \qquad v' \subseteq v \subseteq \mathcal{V}_n}{\begin{array}{c}(v,0) \xrightarrow{\mathit{cl}\ (v',\mathit{cl}):\_} (v, 1) \in \mathit{rul}^{A_{\mathit{Bef}\bullet}} \\ (v,\flat) \xrightarrow{\mathit{cl}\ (v',\alpha):\_} (v, \flat) \in \mathit{rul}^{A_{\mathit{Bef}\bullet}}\end{array}}$$

$A_{\mathit{Bef}\bullet}$ can be computed in time $O(3^n)$: For opening rules, choosing v and v' consists in determining for each variable $x \in \mathcal{V}_n$ whether $x \in v - v'$, $x \in v' - v$ or $x \notin v \cup v'$. Similarly, for closing rules, we have to choose whether $x \in v - v'$, $x \in v'$, or $x \notin v \cup v'$. □

Finally, the relation $C_{2^{\mathcal{V}_n}} \subseteq \mathcal{T}_{2^{\mathcal{V}_n}}$ is the set of trees of $\mathcal{T}_{2^{\mathcal{V}_n}}$ of type $1^{\mathcal{V}_n}$.

Lemma 29. *An dSTA recognizing $C_{2^{\mathcal{V}_n}}$ can be computed in time $O(3^n)$.*

Proof. Here we just have to collect variables in states at opening, and read only variables that have not been seen so far.

$$\mathit{stat}_{\mathrm{e}}^{A_{C_{2^{\mathcal{V}_n}}}} = 2^{\mathcal{V}_n} \qquad \mathit{init}^{A_{C_{2^{\mathcal{V}_n}}}} = \{\emptyset\} \qquad \mathit{fin}^{A_{C_{2^{\mathcal{V}_n}}}} = \{\mathcal{V}_n\} \qquad \mathit{stat}_{\mathrm{n}}^{A_{C_{2^{\mathcal{V}_n}}}} = \{\_\}$$

$$\frac{v, v' \subseteq \mathcal{V}_n \qquad v \cap v' = \emptyset}{v \xrightarrow{\mathit{op}\ v':\_} v \cup v' \in \mathit{rul}^{A_{C_{2^{\mathcal{V}_n}}}}} \qquad \frac{v' \subseteq v \subseteq \mathcal{V}_n}{v \xrightarrow{\mathit{cl}\ v':\_} v \in \mathit{rul}^{A_{C_{2^{\mathcal{V}_n}}}}}$$

The complexity comes from the same argument as Lemma 28. □

7.5.2 Bounded Delay

Our objective is to define the formulas delay_Q and concur_Q in the logic $\mathrm{FO}_\exists[\mathit{Eq}, \mathit{Can}, S, \mathit{Bef}, \mathit{Bef\&Can}]$ preferably without using Bef. Relational structures for interpretation are fixed by a query Q, which maps the relation symbols to the following recognizable relations Can_Q, $\mathit{Bef\&Can}_Q$, and $S_Q = \mathit{dom}(Q)$. All other relation symbols have a fixed interpretation by the relation of the same name.

We start with the definition of the relation $\mathit{Sel}_Q = \{(t, \mathit{ren}^\tau(t), \mathit{ren}^\eta(t)) \mid (\tau, \eta) \in \mathit{sel}_Q(t)\}$ by an FO formula *Sel* with three free variables, such that $\mathit{Sel}_Q = R_{\mathit{Sel}(T_t:\mathcal{T}_\Sigma, T_\tau:\mathcal{T}_{2^{\mathcal{V}_n}}, T_\eta:\mathcal{T}_{\{0,op,cl\}})}(Q)$:

$$\begin{aligned}\mathit{Sel} \;=_{df}\; & S(T_t) \wedge \mathit{Bef}(T_t, T_\tau, T_\eta) \\ & \wedge \forall T'_t \in \mathcal{T}_\Sigma.\, (S(T'_t) \wedge \mathit{Eq}(T_t, T'_t, T_\eta)) \Rightarrow \mathit{Can}(T'_t, T_\tau)\end{aligned}$$

Note that entailment of $\mathit{Can}(T'_t, T_\tau)$ is correct only since we prune trees using *Bef*: if (t', t, η) belongs to relation $R_{\mathit{Eq}(T_t:\mathcal{T}_\Sigma, T'_t:\mathcal{T}_\Sigma, T_\eta:\mathcal{T}_{\{0,op,cl\}})}$ then t and t' may have different domains beyond η. Given dSTAs A and B defining $Q = Q_{A,B}$ we can thus define a dSTA recognizing $\mathit{Sel}_Q(T_t, T_\tau, T_\eta)$. Unfortunately, we cannot construct this dSTA in PTIME yet, since formula *Sel* does not belong to the existential fragments of FO and uses relation *Bef*. Nevertheless, we obtain algorithm for deciding judgments $(\tau, \eta) \in \mathit{sel}_Q(t)$.

We define the relation $\mathit{Delay}_Q = \{(t, \mathit{ren}^\tau(t), \mathit{ren}^\eta(t)) \mid \eta \in \mathit{delay}_Q(t, \tau)\}$ by the following formula of $\mathrm{FO}_\exists[\mathit{Eq}, \mathit{Bef\&Can}, S, \mathit{Can}]$, that expresses that η is an event increasing the delay if the nodes of $\tau \in Q(t)$ are before η in t, and there is a tree t' that equals t until η but with $\tau \notin Q(t')$. The formula has 3 free variables such that $\mathit{Delay}_Q = R_{\mathit{Delay}(T_t:\mathcal{T}_\Sigma, T_\tau:\mathcal{T}_{2^{\mathcal{V}_n}}, \mathcal{T}_{\{0,op,cl\}})}(Q)$.

$$\begin{aligned}\mathit{Delay} \;=_{df}\; & \exists T'_t \in \mathcal{T}_\Sigma.\, S(T_t) \wedge \mathit{Bef\&Can}(T_t, T_\tau, T_\eta) \\ & \wedge S(T'_t) \wedge \mathit{Eq}(T_t, T'_t, T_\eta) \wedge \neg \mathit{Can}(T'_t, T_\tau)\end{aligned}$$

All base relations can be defined by dSTAs of polynomial size when leaving n variable (since we do not need the relation *Bef* here, and by Lemmas 25, 26 and 27). Given deterministic automata A and B, we can thus define a possibly nondeterministic automaton recognizing $\mathit{Delay}_{Q_{A,B}}(T_t, T_\tau, T_\eta)$ in PTIME from A and B. Let $\mathit{2Delay}_Q = \{(t \circledast s_\tau, s_\eta) \mid (t, s_\tau, s_\eta) \in \mathit{Delay}_Q\}$. Both relations are recognized by the same automaton. This relation exactly captures the delay:

$$\mathit{val}(\mathit{2Delay}_Q) = \max_{\tau \in Q(t)} \mathit{delay}_Q(t, \tau)$$

By Proposition 28 we can define automata recognizing relation $\mathit{2Delay}_Q$ in PTIME, so that we can decide bounded delay and k-bounded delay of Q for a fixed k in PTIME by Theorems 13 and 14.

7.5.3 Bounded Concurrency

For concurrency, we proceed in a similar manner.

Proposition 31. *If arity $n \in \mathbb{N}$ is fixed, then for every n-ary query $Q = Q_{A,B}$ defined by dSTAs A and B, we can compute in* PTIME *a possibly nondeterministic STA that recognizes the relation $\mathit{Alive}_Q = \{(t, \mathit{ren}^\tau(t), \mathit{ren}^\eta(t)) \mid (\tau, \eta) \in \mathit{alive}_Q(t)\}$.*

Proof. We define Alive_Q by a formula of $\mathrm{FO}_\exists[S, \mathit{Can}, \mathit{Eq}_\Sigma, \mathit{Eq}_{2^{\mathcal{V}_n}}, C_{2^{\mathcal{V}_n}}, \mathit{Bef}_\bullet]$, such that $\mathit{Alive}_Q = R_{\mathit{Alive}(T_t:\mathcal{T}_\Sigma, T_\tau:\mathcal{T}_{2^{\mathcal{V}_n}}, T_\eta:\mathcal{T}_{\{0,op,cl\}})}(Q)$. Here we use the relation Eq with two different alphabets: Σ and $2^{\mathcal{V}_n}$. The latter permits to express completions of tuples.

$$\begin{aligned}\mathit{Alive}(T_t,T_\tau,T_\eta) =_{df}\ & \exists T'_t \in \mathcal{T}_\Sigma.\ \exists T''_t \in \mathcal{T}_\Sigma.\ \exists T'_\tau \in \mathcal{T}_{2^{\mathcal{V}_n}}.\ \exists T''_\tau \in \mathcal{T}_{2^{\mathcal{V}_n}}.\\ & S(T'_t)\ \wedge\ S(T''_t)\\ & \wedge\ \mathit{Can}_Q(T'_t, T'_\tau)\ \wedge\ \mathit{Eq}_\Sigma(T_t, T'_t, T_\eta) \wedge \mathit{Eq}_{2^{\mathcal{V}_n}}(T_\tau, T'_\tau, T_\eta)\ \wedge\ \mathit{Bef}_\bullet(T_\tau, T_\eta)\\ & \wedge\ \neg\mathit{Can}_Q(T''_t,T''_\tau)\ \wedge\ \mathit{Eq}_\Sigma(T_t,T''_t,T_\eta)\ \wedge\ \mathit{Eq}_{2^{\mathcal{V}_n}}(T_\tau,T''_\tau,T_\eta)\ \wedge\ C_{2^{\mathcal{V}_n}}(T''_\tau)\end{aligned}$$

This formula expresses that τ is alive at η of $t \in \mathcal{T}_\Sigma$ if there exists continuations $t', t'' \in \mathcal{T}_\Sigma$ of t beyond η and two completions τ', τ'' of τ beyond η such that $\tau' \in Q(t')$ but $\tau'' \notin Q(t'')$. $\mathit{Bef}_\bullet$ checks whether $\mathit{latest}(\tau) \preceq \eta$. $C_{2^{\mathcal{V}_n}}$ verifies that T''_τ is canonical, as this is not done by $\neg\mathit{Can}_Q(T''_t,T''_\tau)$. All relations used in the formula are recognizable by automata that can be computed in PTIME by Lemmas 25, 26, 28 and 29, so that an STA for Alive_Q is obtained from Proposition 28 (since A is deterministic). Indeed, this result remains true if B is nondeterministic, since relation symbol S does not occur below negation. □

Note that we cannot integrate the canonicity control for T''_t into the negated relation $\neg\mathit{Can}(T''_t,T''_\tau)$. The deeper problem is that automata A for canonical languages of queries $Q_{A,B}$ do not have a notion of safe states *in the case of trees*, since safety depend also on the current stack content.

Let $\mathit{2Alive}_Q$ be the binary version of Alive_Q, i.e., $\mathit{2Alive}_Q = \{(t \circledast s_\eta, s_\tau) \mid (t, s_\eta, s_\tau) \in \mathit{Alive}_Q\}$, then:

$$\mathit{val}(\mathit{2Alive}_Q) = \max_{t \in \mathit{dom}(Q)} \mathit{concur}_Q(t)$$

We can recognize $\mathit{2Alive}_Q$ with the same automaton as Alive_Q, which can be computed in PTIME for fixed n from A and B by Proposition 31. Hence we can decide bounded and k-bounded concurrency of Q for fixed n and k in PTIME by Theorems 13 and 14. The cost of the automaton construction is in $O(p(|\Sigma|, |A|, |B|) \cdot (2^n)^4 \cdot (3^n)^2)$ for some polynomial p: building the automaton for $\mathit{Eq}_{2^{\mathcal{V}_n}}$ is in $O((2^n)^2)$ by Lemma 25, and the automata for $\mathit{Bef}_\bullet$ and $C_{2^{\mathcal{V}_n}}$ are built in $O(3^n)$ by Lemmas 28 and 29. A lower complexity may be obtained by more ad hoc constructions, for instance by directly computing an automaton for Alive_Q.

7.5.4 Discussion of Direct Construction

We end this section by pointing out an alternative (and more direct) construction, that computes in time $O(p(|\Sigma|, |A|, |B|) \cdot (2^n)^2)$ (for some polynomial p) an STA

recognizing $Alive_Q$. In Chapter 5, we explained how to compute a dSTA $E(A)$ recognizing $L(A)$, and such that each state is either safe or unsafe for selection (and respectively for rejection). This cannot be done for A, as the safety condition depends on the current configuration, which contains a stack content. This comes however at a cost: each state of $E(A)$ includes a set of safe states, and thus the size of $E(A)$ is in $O(2^{|A|})$.

To avoid this blowup, we use non-determinism. When building $E(A)$, a new set of safe states is computed for each opening rule. Instead of computing this set, we guess non-deterministically a state that is unsafe for selection and a state that is unsafe for rejection. Hence states of A_{Alive} are 3-tuples of states of A: one state for the run of A and two unsafe states. The computation of unsafe states follows the same line as the computation of safe states for $E(A)$. We just have to replace a universal quantification on continuations (they all have to be safe) by an existential quantification (one must fail, to be unsafe for selection).

While avoiding a blowup in the size of A, we still have to make it complete, which requires time in $O(|\Sigma| \cdot (2^n)^2)$. The completion is needed, as there must be an accepting run of A_{Alive} when we reach an unsafe state for selection at η (if the second state of the pair was also unsafe for rejection). Note that this alternative construction requires that the automaton B recognizing the schema language is deterministic. This is not the case for the construction using recognizable relations.

7.6 Conclusion

In this chapter, we proved that deciding whether a query defined by dSTAs has bounded (resp. k-bounded) delay and concurrency can be performed in polynomial time, for a fixed k. We chose to focus on measures of delay and concurrency that were motivated by query answering in a streaming manner. Some extensions of these measures could be also investigated, especially for the delay. For instance we studied the delay for selecting a tuple, but we could also study the delay for rejecting a candidate tuple. This measure is close to concurrency, as bounded delay for rejection implies bounded concurrency, whereas bounded delay for selection does not (for n-ary queries).

We also chose to measure the delay from the point where the candidate tuple gets complete, as it cannot be output before. We could define the i-th delay like in our definition, but starting to count when i components of the tuple are filled. Hence n-th delay would be the delay studied in this chapter. This would make sense if we want to decide whether all completions of a partial tuple will succeed, and in this case output it. Then the completion with any incoming node could be performed by a parallel process.

Another variant for the i-th delay is to measure the number of events between completing i components and completing $i+1$ components of the candidate tuples. If all these delays are bounded, then the query has bounded delay, according to the definition studied in this chapter. This would give intermediate measures of bounded delay. For instance, we could characterize queries for which components of candidates are quickly filled, except one component for which the delay may be unbounded. This could help designing streamable queries.

In terms of improvements, we would like to replace the reduction to the bounded valuedness of tree transducers to a more direct construction. Indeed, tree transducers are more powerful than binary recognizable relations, so we can hope for more efficient algorithms. This requires however to consider two kinds of non-determinism inside the automaton recoginzing the query: the usual non-determinism (on runs of the automaton) and the non-determinism on the second component of the binary relation. Another open question is whether a restriction on shallow trees could lead to more efficient algorithms.

Chapter 8

Conclusion

8.1 Main Results

The work presented in this manuscript focused on XML data, and more specifically to the query answering over XML streams. We addressed two kinds of queries. The first one is XPath, a W3C standard based on a navigational language. The second one is tree automata, a tool originating from language theory, that we use here as query definition language. Usually, XML data come with a schema that describes the structure of valid XML documents. We took schemas into account in our framework, as they can improve the efficiency of query answering algorithms. All query classes that we studied allow the definition of n-ary queries, i.e., queries that select n-tuples of nodes, instead of simple nodes.

We started this dissertation with a description of our framework for query answering on streams in Chapter 3. To establish a clear definition, and get a precise complexity measure, we introduced Streaming Random Access Machines (SRAMs). These are RAMs with some registers, a working memory and two tapes: a read-only input tape and a write-only output tape. Then we introduced a measure for the streamability of queries. A query is said streamable if there is an algorithm computing it, that uses a PTIME preprocessing, and polynomial space and time for processing each event of the stream. These complexity measures are in the size of the query, but constant in the size of the tree. By relaxing these strong requirements, we defined a hierarchy of m-streamable query classes, for $m \in \mathbb{N}_0$. Then we studied the streamability of queries defined by XPath and tree automata, the two query classes studied in this manuscript. We proved that both are not streamable, even at low levels of our hierarchy. This motivated the investigation of streamable fragments.

For tree automata, we defined Streaming Tree Automata (STAs), a model that evaluates trees according to a pre-order traversal. This corresponds to the way

a tree is read when its corresponding XML document is accessed in a streaming mode. In Chapter 4, we studied the links between STAs and other automata models: models that also evaluate in pre-order (nested word automata, visibly pushdown automata and pushdown forest automata) and standard models that evaluate in a bottom-up or top-down manner. In particular, deterministic STAs (dSTAs) can be obtained in PTIME from all other models. In Chapter 5, we proved that dSTAs are m-streamable on shallow trees for all $m \in \mathbb{N}_0$. To get this positive result, we introduced Earliest Query Answering (EQA). An EQA algorithm outputs each answer at the earliest time point where it can be decided that it is selected by the query, whatever the continuation of the stream is. This algorithm also discards candidates that will not be selected in any continuation, at the earliest time point. We study the complexity of such algorithms, and establish lower bounds. These bounds are of great interest, as any streaming query answering algorithm with optimal memory consumption has to be an EQA algorithm, and thus these lower bounds indicate how much time is needed to reach optimal space complexity. The m-streamability of dSTAs is shown by building an EQA algorithm for queries defined by dSTAs, that uses polynomial per-event space and time, for each candidate that needs to be buffered.

For queries defined by XPath expressions, we proposed k-Downward XPath (for $k \in \mathbb{N}$), a set of fragments suitable to streaming evaluation. k-Downward XPath is m-streamable for all $m \in \mathbb{N}_0$. It allows only downward axes *ch* and *ch**, and restricts the inherent non-determinism of XPath, so that k-Downward XPath expressions can be translated in PTIME to equivalent dSTAs. The positive streamability results were obtained by reduction to streamability of dSTAs, as previously described. Our translation to dSTAs allows us to apply all our algorithms for dSTAs on k-Downward XPath expressions, in particular the EQA algorithm, and the decision procedures described in the sequel.

Finally, we established that deciding bounded (and k-bounded) delay and concurrency of queries defined by dSTAs can be decided in PTIME. The delay of a monadic query is the maximal number of events between reading a selected node, and the earliest event where it can be decided that it will be selected in any continuation of the stream. For n-ary queries, we start measuring the delay when the tuple is filled. Hence having k-bounded delay ensures that once a candidate is complete, we have to way at most k events before being able to output it. The concurrency is the number of simultaneously alive candidates, i.e. candidates that have to be buffered, as their selection or rejection cannot be decided yet. Both results were established using properties of recognizable relations over unranked trees, for which we proved that the bounded valuedness can be decided in PTIME for a given k, even from non-deterministic automata.

8.2 Perspectives

Throughout the dissertation, we studied the scalability of query classes through our notion of streamability. We proved non-streamability for some classes (XPath, non-deterministic tree automata) and also m-streamability for some others, for all $m \in \mathbb{N}_0$ (k-Downward XPath, and dSTAs). However, we did not provide a method to effectively compute the degree of streamability of a query class, when it is in-between. In particular, it would be interesting to find characterizations that are equivalent to m-streamability. Moreover, our computational model implies a memory lower bound for all queries (see Proposition 7). Some results by Bar-Yossef et al. [BYFJ05] prove that this bound is a real lower bound for any query answering algorithm for some fragment of XPath. It is still open whether this also holds for other XPath fragments, and for queries defined by tree automata.

In Chapter 6, we have seen that translating k-Downward XPath to dSTAs proved the m-streamability of k-Downward XPath, for all $m \in \mathbb{N}_0$. An open question (which was also our working hypothesis) is whether query classes for which a PTIME translation to dSTAs exist are exactly query classes that are m-streamable for all $m \in \mathbb{N}_0$. This would prove that dSTAs are the good model for defining streamable queries. Another interesting characterization of streamability could also exist at the level of logics, as proposed recently by Ley and Benedikt [LB09]. In particular, it is known that FO formulas can only describe local properties. This may restrict the number of simultaneous candidates, and thus lead to streamable query classes. However, when allowed moves (i.e. predicates) are not along the document order, this fails. For instance allowing transitive closure in axes like ch^* allows jumps in the tree, and thus moves with unbounded delay. Even the next-sibling axis *ns* is problematic, as the number of events between the opening of two direct siblings can be unbounded, even on shallow trees. All streamable classes studied in this dissertation have a semantic restriction on the depth of trees, i.e. only consider shallow trees. Then a question is whether we could use this fact to get better algorithms. For instance we could translate tree automata to word automata (recognizing the words of tags) on the fly, and use more efficient algorithms for words. Moreover, we only focused on queries that only take the structure of the tree into account, not the textual data.

The framework adopted in this dissertation may be extended in several ways. First, we could allow multiple scans over the XML stream, instead of a single pass. This makes sense for stored data that can be read several times. This was studied by Grohe, Koch and Schweikardt [GKS07] for XPath, but not for queries by automata. It would be also interesting to study how several queries can be simultaneously computed on several XML streams. The challenge here is to find a data structure for the compact representation of the set of candidate tuples. This question is also relevant for our EQA algorithm for dSTAs, where we did not ad-

dress this problem. It was studied for instance by Meuss et al. in [MSB01], but outside the scope of a streaming evaluation. Another alternative framework for XML streams is the use of indexed streams, where one stream is defined for each label of the alphabet, and in each stream, elements are accessed in document order. This has been recently investigated by Shalem and Bar-Yossef, for the restricted case of tree patterns [SBY08]. More generally, this raises the question of XML serialization. It could be interesting to allow more flexible forms of serialization, not only the document order. The way XML documents (and their schemas) are generated usually ignores which queries will have to be evaluated on these documents. Hence the information may be stored in a different order than what is needed for the evaluation of queries. To solve this problem, a solution could be to distinguish between the DOM representation of an XML document and its serialization, by serializing it according to some information on potential queries asked on this document.

Concerning the earliest query answering algorithms studied in Chapter 5, the goal was to prove lower memory bounds. As a consequence, the tradeoff between space and time complexity is here on the extreme side of optimal space consumption, at any time cost. A way to relax this requirement is to find heuristics, as investigated by Benedikt et al. [BJLW08] for approximating the earliest rejection of candidates. Other results are known for approximate query answering, as those established by De Rougemont et al. [CJdR08, dRV08]. Approximate validation of XML streams has been investigated by Thomo et al. in [TVY08], and Schewe et al. in [STW08]. Another way to relax the earliest decision requirement is to postpone these decisions (selection or rejection) to a time point where we are sure that enough information has been read. This is a common solution in existing algorithms. For instance for fragments of XPath allowing only downward moves and tests, the decision for selecting a node is usually done when closing it. It could be interesting to try to improve this, for instance by considering schema information.

Query answering is a first step towards the evaluation of transformations. Hence a natural extension of our work is to take XQuery FLOWR expressions into account. These are for-loops with variables, that can be nested, and also select tuples of nodes. The next step is to produce the output XML document progressively. This will create new difficulties, as once more we will have to decide whether some part can be output because it will not change in any continuation of the input stream. Transformation languages contain some other features like aggregators, and their streaming evaluation also has to be studied. XProc proposes to define transformations through XML pipelines. This language allows to separate regions of the XML tree where a transformation (defined for instance in XQuery or XSLT) occurs, and thus avoids to buffer too much information. This is why this language looks more suitable to a streaming evaluation than XQuery transformations on full documents.

Chapter 9

Résumé

9.1 Contexte

Le format XML, introduit il y a dix ans, s'est imposé comme le standard pour les applications orientées Web et le traitement des documents [BPSM+08]. Emanant de SGML, XML définit des documents semi-structurés, modélisés par des arbres. La syntaxe d'un document XML est une suite de balises bien imbriquées, dont certaines contiennent des données textuelles. Ceci diffère des bases de données relationnelles, où les données sont stockées dans des tables. Avec XML sont apparus des langages de schémas comme les DTDs (Document Type Definition), XML Schema ou Relax NG. Un schéma définit la structure attendue des documents XML utilisés au sein d'une application donnée.

Considérons par exemple le document XML représenté dans la figure 9.1(a). Ce document contient des données géospatiales concernant deux villes, et est modélisé par l'arbre représenté dans la figure 9.2. Un schéma pour ce document est présenté dans la figure 9.1(b).

Le premier type de traitement des documents XML est la *validation* d'un document par rapport à un schéma donné. Ceci est nécessaire aux applications manipulant des données XML, afin de de s'assurer de leur conformité envers le schéma souhaité. Le second type de traitement consiste à répondre aux requêtes, c'est-à-dire à trouver les nœuds d'un document XML sélectionnés par une requête. Il s'agit d'une étape de base pour récupérer des informations dans un document XML. Dans notre exemple il peut être intéressant de sélectionner les triplets (`nom`, `lat`, `lon`). Le *filtrage* est un cas particulier de réponse aux requêtes, où il suffit de déterminer si un document XML possède une solution par rapport à une requête. Le troisième type de traitement est la *transformation* de documents XML, elle-même souvent basée sur une notion de requêtes. Les transfor-

```
<geo>
  <point>
    <nom>Lille</nom>
    <lat>50.63050</lat>
    <lon>3.07063</lon>
  </point>
  <point>
    <nom>Hellemmes</nom>
    <lat>50.62746</lat>
    <lon>3.10853</lon>
  </point>
</geo>
```

(a) Document XML.

$$
\begin{aligned}
\texttt{geo} &\rightarrow \texttt{point}^* \\
\texttt{point} &\rightarrow (\texttt{nom}, \texttt{lat}?, \texttt{lon}?) \\
\texttt{nom} &\rightarrow \text{\#PCDATA} \\
\texttt{lat} &\rightarrow \text{\#PCDATA} \\
\texttt{lon} &\rightarrow \text{\#PCDATA}
\end{aligned}
$$

(b) Schéma défini par une DTD.

Figure 9.1: Fichier XML contenant des données géospatiales, conforme à une DTD.

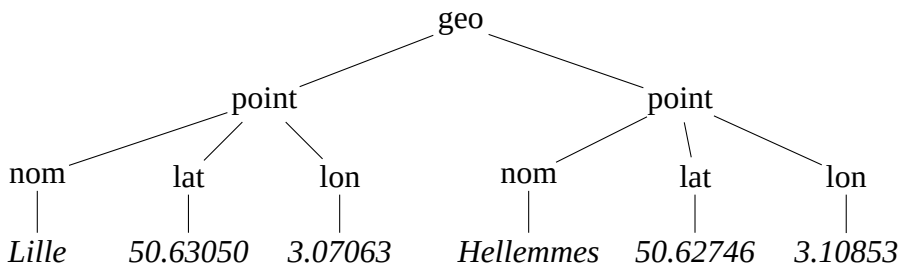


Figure 9.2: Représentation arborescente du fichier XML de la figure 9.1(a).

mations possèdent beaucoup d'applications dans le cadre des documents XML. Par exemple l'échange de données consiste à transformer un document conforme à un schéma, en un document conforme à un autre schéma. La transformation de données désigne l'ensemble des transformations d'un document XML en un autre. Un autre exemple fréquent est la transformation des documents XML en pages Web, en utilisant des feuilles de style XSLT.

Toutes ces types de traitement peuvent être effectués selon différents modes. Le premier est l'*évaluation en mémoire centrale*. Dans ce cas, le document XML est entièrement chargé en mémoire centrale, puis traité. La sortie est produite uniquement lorsque l'ensemble des solutions est calculé. L'un des inconvénients de cette méthode est une consommation mémoire importante. Un autre inconvénient est de devoir attendre la fin du traitement pour produire les sorties, alors que souvent certaines sont connues avant. Une autre approche permettant de

résoudre cet inconvénient est l'*énumération* des solutions. Cela consiste à sortir, après une phase de précalcul, chaque solution, l'une après l'autre, avec un délai raisonnable entre deux solutions consécutives. Enfin, le mode d'évaluation *en flux* (*streaming*) impose davantage de restrictions sur la consommation mémoire. Dans ce mode, le document XML est lu en une seule passe, de la première balise à la dernière. Cet ordre est appelé ordre du document. La sortie est également produite en flux : lorsqu'une solution est trouvée, ou qu'une partie du document de sortie est produite, elle est envoyée sur un périphérique de sortie. L'objectif d'une évaluation en flux est d'utiliser moins de ressources mémoire, en ne stoquant que l'information nécessaire. Le stockage est nécessaire lorsque la sortie dépend de la suite du flux d'entrée. Le but est de pouvoir traiter des documents ne pouvant être chargés en mémoire centrale, ou de traiter à la volée des flux XML provenant d'un réseau.

Plusieurs standards ont été mis en place pour les différents types de traitements évoqués ci-dessus. Nous avons déjà illustré les langages de schéma par les DTDs, définies au sein du standard XML [BPSM+08]. XML Schema [FW04] est une extension des DTDs permettant par exemple de caractériser plus précisément le contenu des données textuelles. De plus, les schémas définis en XML Schema sont eux-mêmes des documents XML, à la différence des DTDs. Relax NG [vdV03] décrit la structure des arbres valides, et délègue la spécification des données textuelles valides à XML Schema.

XPath [CD99] est le standard pour la sélection de nœuds dans les documents XML. XPath est basé sur la description des chemins, par des suites d'étapes à suivre jusqu'à atteindre les nœuds sélectionnés. XPath permet également d'ajouter des filtres à chaque étape. Un filtre est une combinaison booléenne d'expressions de chemins, et est satisfait si un nœud satisfait cette combinaison. Il est également possible de tester le contenu textuel des nœuds. XPath est un langage de requête central, utilisé comme mécanisme de sélection de nœuds dans de nombreux autres langages, comme XPointer [DMJ01], un standard pour la sélection de fragments dans les documents XML.

XPath est également utilisé par les deux langages de transformation XQuery [BCF+07] et XSLT [Cla99]. XQuery est un langage impératif utilisant des boucles *for* pour sélectionner des tuples de nœuds. Ceux-ci sont ensuite insérés dans un contexte XML pour produire un document XML de sortie. XSLT est plus proche de la programmation fonctionnelle. Une feuille de style XSLT est composée de patrons, activés pour les nœuds satisfaisant l'expression XPath.

XProc [WMT09] propose de combiner tous ces standards grâce à un langage de pipelines. Alors que XPath, XQuery et XSLT n'étaient pas conçus pour une évaluation en flux, XProc permet de définir des parties de l'arbre où opèrent la sélection et la transformation. Ainsi, les difficultés inhérentes à l'évaluation en

flux sont circonscrites à certaines régions. Comme évoqué dans ce manuscrit, d'autres langages, comme STX [BBC02], ont été conçus spécifiquement pour une évaluation en flux, mais aucun standard n'a été adopté.

Les automates finis de mots [HU79] opèrent sur les mots en un seul passage, afin de décider de leur appartenance au langage de l'automate. Ainsi, ils évaluent naturellement les mots en flux. Ces objets ont été étudiés de longue date, et bénéficient de liens intéressants avec la logique et la théorie des langages. Les documents XML sont modélisés par des arbres, et non par des mots. Cependant, les documents XML de base sont des linéarisations de ces arbres : un document XML est une suite de balises (un flux XML), et donc un mot. Ici les balises sont bien imbriquées, et reflètent la structure d'arbre. Les automates de mots sont incapables de prendre en compte cette relation d'imbrication. Nous avons donc besoin d'un modèle d'automates plus puissant pour traiter les flux XML.

Les automates d'arbres [CDG+07] fournissent un cadre pour la définition et l'étude des traitements XML. Des relations directes avec la logique et la théorie des langages d'arbres ont été également établies au travers de nombreux travaux. En particulier, ils représentent un cadre algébrique pour les bases de données XML, de la même manière que l'algèbre relationnelle pour les bases de données relationnelles. Il a été montré que les automates d'arbres capturent tous les langages de schémas standards, et la traduction d'un schéma en automate d'arbre est relativement simple [MLM01]. Les automates d'arbres ont également été proposés comme mécanisme de définition de requêtes dans les arbres [NS02, Koc03, BS04, CNT04]. Les expressions XPath peuvent également être traduites en automates d'arbres, mais cette fois la traduction n'est pas triviale. La validation et le traitement des requêtes ont également été étudiés pour les automates d'arbres. Les transformations sont définies par des transducteurs d'arbres. Par rapport aux automates d'arbres, ils permettent de produire une sortie tout en lisant l'entrée.

9.2 Motivations

Dans ce manuscrit, nous étudions les algorithmes de réponse aux requêtes, utilisant une évaluation en flux, pour des requêtes définies par des expressions XPath et des automates d'arbres. L'évaluation en flux est désormais un défi majeur pour le traitement des requêtes XPath. Michael Kay, le concepteur de Saxon (le moteur de référence pour XQuery) déclarait récemment [Kay09] :

> Les capacités de traitement en flux [de Saxon] sont désormais l'une des principales raisons pour lesquelles les gens achètent le produit.

entrée	a	b	a	a	b	b	b	a	b	b
mémoire	1	1	3	4	4			8	8	
sortie						4				8

Figure 9.3: Evaluation en flux pour la sélection des positions a suivies par $b \cdot b$.

Le traitement en flux des documents XML est étudié depuis longtemps. Nous illustrons ce mode d'évaluation et les concepts afférents par une requête sur les mots de l'alphabet $\{a, b\}$. Considérons la requête qui sélectionne les positions étiquetées par a, et directement suivies par bb. Par exemple, sur le mot $abaabbbabb$, cette requête sélectionne les positions 4 et 8, comme indiqué dans la figure 9.3. Toutes les positions étiquetées par b peuvent immédiatement être écartées. Pour les positions étiquetées par a, la sélection ou le rejet d'une position candidate ne peuvent pas être décidés immédiatement. Les positions suivies par a (comme la position 3) peuvent être rejetées après une étape, et celles suivies par $b \cdot a$ (comme 1) après deux. Cette requête peut être évaluée avec une fenêtre (*sliding window*) de longueur 3, et nécessite de mémoriser au plus un seul candidat à la fois. Nous appelons *délai* la taille minimale de la fenêtre, et *concurrence* [BYFJ05] le nombre minimal de candidats simultanément vivants. Un candidat est *vivant* à un certain moment, s'il existe une continuation du flux permettant sa sélection, et une autre permettant son rejet. Ainsi les candidats vivants nécessitent d'être mémorisés. Il est souvent facile de définir des requêtes ayant une concurrence élevée, par exemple ici en permettant que $b \cdot b$ apparaisse après a, mais pas immédiatement. Les schémas peuvent permettre de réduire la quantité de données à mémoriser. Par exemple supposons que tous les mots valides sont tels qu'une fois que trois b successifs sont apparus, toute position a est suivie par $b \cdot b$. Dans ce cas, toutes les positions étiquetées par a apparaissant après trois b successifs pevent être immédiatement sélectionnées. Par exemple dans notre cas, la position 8 peut être sortie à la position 8 au lieu de la position 10.

Dès les premiers travaux, les algorithmes d'évaluation en flux ont montré de meilleurs performances, mais ne permettaient de n'évaluer que des fragments restreints des langages de requêtes. De nombreuses difficultés liées à ce mode d'évaluation ont été identifiées. Pour la validation [SV02], un premier obstacle est la nature récursive des documents XML. Le traitement de documents récursifs nécessite de stocker dans une pile des informations à propos des ancêtres des nœuds. Ainsi la mémoire peut être bornée par la profondeur de l'arbre, mais ne peut pas être bornée indépendamment pour tous les arbres. Les langages de requête comme XPath sont, de manière inhérente, non déterministes [PC05], à la différence des langages de schémas. Par exemple, XPath permet de parcourir l'arbre suivant l'axe *descendant*. En partant d'un nœud, cela corre-

spond à sélectionner tous ses descendants, et donc génère de nombreux candidats pour l'étape suivante. Parmi ces candidats, certains auront besoin d'être stockés, puisqu'ils peuvent avoir besoin d'informations supplémentaires pour déterminer s'ils satisfont la requête. Ces difficultés apparaissent déjà pour le filtrage de documents XML par des expressions XPath [AF00]. De plus, XPath permet le branchement, via les filtres et les conjonctions au sein des filtres. Cela augmente souvent la complexité des algorithmes. Les transformations apportent des problèmes supplémentaires pour l'évaluation en flux [FHM+05, Mic07]. C'est typiquement le cas pour les opérateurs manipulant les positions parmi les éléments sélectionnés, par exemple en cherchant le dernier élément sélectionné, ou pour trier ces éléments.

Par rapport à ces aspects bloquants, des bornes inférieures pour la mémoire ont été établies pour ces différents traitements. Pour les requêtes, la notion centrale est la concurrence, précédemment introduite. Il a été montré [BYFJ05] que la concurrence est une borne inférieure pour la mémoire, lors du traitement des requêtes XPath appartenant à un certain fragment. Cela amène à se poser la question suivante : peut-on atteindre cette borne ? Cette question peut être décomposée en plusieurs variantes. Tout d'abord, ce résultat se généralise-t-il à d'autres classes de requêtes ? Il serait également intéressant de savoir si cette borne inférieure est proche de la borne supérieure, c'est-à-dire s'il existe des algorithmes dont la consommation mémoire soit proche de cette borne inférieure. Quel est le coût en temps de calcul pour atteindre de telles bornes ? En d'autres termes, ces algorithmes nécessitent-ils des temps de calcul importants pour décider de la sélection ou du rejet des candidats ? Comment ces coûts varient-ils d'une classe de requêtes à l'autre ? Existe-t-il des classes de requêtes pour lesquelles des algorithmes efficaces existent ? Ces classes sont-elles caractérisées pour une certaine propriété ? Les classes ayant une concurrence non bornée peuvent-elles être traitées efficacement ? Quelles requêtes nécessitent peu de mémorisation (même si cette mémorisation ne peut être bornée) ? Ces questions motivent la définition d'une mesure plus fine que la concurrence : la *streamabilité* d'une requête, i.e. une notion mesurant à quel point une requête est adaptée à une évaluation en flux. La concurrence établit une première frontière entre les requêtes ayant une concurrence bornée (et pouvant ainsi être évaluées avec une mémoire bornée sur des arbres de profondeur bornée) et les autres. Mais les questions ci-dessus justifient la définition d'une notion plus fine de streamabilité.

Nous nous intéressons aux requêtes n-aires, pour $n \geq 0$. Celles-ci sélectionnent des n-uplets de nœuds dans les arbres. Le cas $n = 0$ correspond aux requêtes booléennes, qui peuvent uniquement distinguer les arbres sélectionnant le tuple vide des autres arbres. Ainsi les requêtes booléennes définissent des langages d'arbres, et sont utilisées pour filtrer les arbres satis-

faisant certaines contraintes. Pour $n = 1$, nous obtenons les requêtes monadiques, qui sélectionnent dans chaque arbre un sous-ensemble de ses nœuds. La sélection de n-uplets de nœuds est une opération centrale dans les langages de transformation. Dans XPath 2.0 et XQuery, cette opération est effectuée via des boucles *pour* imbriquées, appelées expressions FLOWR. XPath 1.0 définit uniquement des requêtes monadiques. En ajoutant des variables, nous permettons à XPath 1.0 de définir des requêtes n-aires. Par rapport aux expressions FLOWR, cela donne plus de flexibilité en terme d'évaluation, et peut compliquer la tâche de nos algorithmes. Les expressions FLOWR sont des instructions de plus bas niveau, permettant au développeur de définir des requêtes adaptées à une évaluation en flux ou pas. Pour les requêtes par automates, les requêtes n-aires sont définies par des langages d'arbres annotés.

Etat de l'art Atteindre la borne inférieure en terme de consommation mémoire a un coût très important en temps. Benedikt et al. [BJLW08] montrent par exemple que pour XPath avec DTDs, pouvoir rejeter les candidats ayant échoué au plus tôt, avec un algorithme construit en temps polynomial par rapport à la taille de la requête, et utilisant un temps polynomial (par rapport à la requête) à chaque événement du flux, est équivalent à PTIME = PSPACE.

Berlea [Ber06, Ber07] étudie les *requêtes régulières d'arbres*, définies par des grammaires d'arbres. Pour cette classe de requêtes, Berlea propose un algorithme basé sur les automates d'arbres, utilisant un espace mémoire optimal en terme de nombre de candidats, tout en traitant chaque événement en temps et espace polynomial, pour chaque candidat. Cependant, cette classe de requêtes suppose un alphabet infini, à la différence des documents XML. La taille infinie de alphabet simplifie grandement le fait de pouvoir sélectionner ou rejeter les candidats au plus tôt.

Certains algorithmes ont été proposés pour l'évaluation en flux de XPath. Pour les axes vers le bas (descendants), nous pouvons mentionner les travaux de Bar-Yossef et al. [BYFJ05, BYFJ07], Ramanan [Ram05, Ram09], et Gou and Chirkova [GC07a]. Les algorithmes de Barton et al. [BCG$^+$03] et de Wu et Theodoratos [WT08] autorisent les axes vers le haut (ancêtres) et vers le bas. Olteanu et al. [OMFB02, OKB03, Olt07b] prouvent que Forward XPath, le fragment de XPath où seuls les axes respectant l'ordre du document sont autorisés, est aussi expressif que XPath (en terme de capacités navigationnelles). Ils proposent *SPEX*, un algorithme efficace basé sur les réseaux de transducteurs, qui évaluent les expressions Forward XPath. Nizar et Kumar [NK08] définissent un algorithme pour les expressions Forward XPath où aucune négation n'apparaît. Récemment, ils étendent cet algorithme aux axes inverses [NK09]. Benedikt et Jeffrey [BJ07] étudient des logiques équivalentes à la partie navigationnelle de

XPath, et déterminent si elles conviennent à une évaluation en flux. Ils identifient des fragments utilisant des modalités vers le bas et dans l'ordre inverse du document, sans négation, de telle sorte que la sélection d'un nœud peut être décidée lors de son ouverture ou de sa fermeture. Pour ces fragments, ils montrent que des algorithmes en temps et espace polynomiaux par événement existent. Benedikt et al. [BJLW08] étudient le filtrage des flux XML par des contraintes XPath, et proposent une heuristique pour la détection au plus tôt des violations de contraintes. Tous ces algorithmes pour l'évaluation de XPath sur des flux XML n'atteignent pas une consommation mémoire optimale, et stockent inutilement des candidats (ou des correspondances partielles) dans certains cas. Ley et Benedikt et al. [LB09] étudient l'existence d'extensions de XPath ayant l'expressivité de la logique du premier ordre, et n'utilisant que des axes compatibles avec l'ordre du document. Ils prouvent que les extensions ayant l'expressivité du premier ordre lorsque tous les axes sont permis ne suffisent pas lorsqu'elles sont restreintes aux axes compatibles avec l'ordre du document.

D'autres bornes inférieures ont été établies, indépendemment de la concurrence. Bar-Yossef et al. [BYFJ04, BYFJ07] prouvent trois bornes inférieures pour des fragments de XPath. La première est la *taille de la frontière de la requête*, c'est-à-dire le nombre maximal de frères des ancêtres d'un nœud, dans la représentation arborescente de la requête. La seconde est la *profondeur de récursion* du document, ce qui correspond au nombre maximal d'ancêtres ayant la même étiquette. La troisième est le logarithme de la *profondeur* de l'arbre. Grohe, Koch et Schweikardt [GKS07], en étudiant des machines de Turing modélisant l'évaluation en flux avec plusieurs passes, montrent que pour la partie navigationnelle de XPath, la *profondeur* de l'arbre est une borne inférieure.

9.3 Contributions

Nous présentons à présent nos contributions. Tout au long du manuscrit, nous considérons les requêtes n-aires, i.e., les requêtes qui sélectionnent des n-uplets de nœuds, au lieu de simples nœuds, comme défini dans XPath 2.0. De plus, nous essayons toujours de prendre les schémas en considération, afin d'améliorer le traitement des flux, puisque les schémas sont souvent disponibles dans les applications concrètes.

Streamabilité Nous commençons par définir un modèle de calcul pour l'évaluation des requêtes en flux : les *Streaming Random Access Machines* (SRAMs). Puis nous introduisons notre notion de *streamabilité*. Nous avions précédemment constaté qu'une telle notion manquait. En raison de l'absence de telles définitions formelles, plusieurs publications présentent des erreurs dans

l'analyse de complexité en espace. De manière simplifiée, pour un entier naturel m, ou pour $m = \infty$, une requête est m-streamable si elle peut être calculée en utilisant un temps et un espace polynomial sur tous les arbres pour lesquels la concurrence de la requête est inférieure à m. Cela introduit une hiérarchie de classes de requêtes. Etre m-streamable avec une valeur élevée pour m est souhaitable, et signifie que les arbres d'entrée entrainant une concurrence inférieure à m peuvent être traités efficacement. Les requêtes ∞-streamables utilisent toujours un temps et un espace polynomial par événement, indépendamment de la concurrence. Nous étudions les relations entre les classes de requêtes ∞-streamables, et les classes de requêtes m-streamable pour tout $m \in \mathbb{N}_0$. Ces dernières doivent avoir une concurrence polynomialement bornée pour être ∞-streamables (pour les requêtes monadiques). Nous étudions la dureté de décider si une classe de requête a une concurrence bornée, ou une concurrence polynomialement bornée. Pour Forward XPath, ces problèmes sont coNP-durs. Nous montrons qu'être 1-streamable a pour conséquence l'existence d'un test d'universalité polynomial sur la classe de requêtes, dès que cette classe vérifie certaines propriétés. Comme l'universalité de Forward XPath est coNP-dure, Forward XPath n'est pas 1-streamable, et donc n'est pas m-streamable, pour tout $m \in \mathbb{N} \cup \{\infty\}$.

Streaming Tree Automata Nous définissons les *Streaming Tree Automata* (STAs), un modèle d'automates évaluant les arbres dans l'ordre du document. Cela correspond exactement à l'ordre d'évaluation du flux XML correspondant. Nous établissons les correspondances entre ce modèle et les autres modèles évaluant dans l'ordre du document, mais sur d'autres structures : les *pushdown forest automata* [NS98], les *visibly pushdown automata* [AM04] et les *nested word automata* [Alu07]. Nous montrons également comment les DTDs peuvent être traduites en STAs, ainsi que les relations entre STAs et les automates d'arbres standard (opérant vers le haut ou vers le bas). Les requêtes définies par des STAs déterministes (dSTAs) sont streamables, dès lors que les arbres ont une profondeur bornée. Nous le prouvons en élaborant un algorithme évaluant les requêtes au plus tôt pour les requêtes définies par dSTAs.

Traitement des Requêtes au plus tôt pour les Streaming Tree Automata Les algorithmes permettant de répondre aux requêtes *au plus tôt* ont la propriété de sortir les réponses aux requêtes dès qu'assez d'informations ont été lues pour assurer la sélection d'une solution, quelle que soit la suite du flux. De manière duale, tous les candidats rejetés sont éliminés dès qu'il est certain qu'aucune suite du flux ne permettra de sélectionner ce candidat (une propriété nommée *fast-fail* dans [BJLW08]). Ce cadre de travail, bien que n'ayant jamais été défini formellement, trouve son origine dans les travaux de Bar-Yossef et al. [BYFJ05] et de

Berlea [Ber06]. Nous proposons une telle définition formelle.

Cette capacité à répondre aux requêtes au plus tôt est requise par tout algorithme ayant une consommation mémoire optimale. Dans le cas contraire, cela signifierait qu'à un certain moment un candidat est inutilement stocké. Cependant, le fait de pouvoir répondre au plus tôt a souvent un coût important en temps de calcul. Pour les requêtes XPath, nous montrons qu'il est coNP-dur de décider si le préfixe d'un flux assure la sélection d'un candidat donné. Pour les requêtes définies par dSTAs, le problème devient traitable, et notre algorithme de réponse au plus tôt fonctionne en temps polynomial, pour une arité n donnée. Ceci fait des dSTAs un modèle robuste pour définir des requêtes adaptées à une évaluation en flux. Notre hypothèse de travail est que toute classe de requête streamable peut être traduite en temps polynomial vers les dSTAs. C'est le cas par exemple pour le fragment de XPath défini ci-après, pour lequel nous fournissons une telle traduction, prouvant ainsi sa streamabilité.

XPath Nous étudions ensuite la streamabilité de XPath plus en détail. Nous identifions une hiérarchie, nommée k-Downward XPath, ayant pour propriété d'être m-streamable pour tout $m \geq 0$. La propriété fondamentale ici est que k-Downward XPath permet de n'avoir au plus qu'un seul candidat simultanément, pour toutes les étapes de chaque branche de l'expression XPath. Pour obtenir cette propriété, nous combinons des restrictions syntaxiques (sur la requête) et sémantiques (sur le schéma). k-Downward XPath est un fragment expressif, par le fait qu'il autorise la négation, le branchement (conjonction et disjonction), ainsi que les axes vers le bas (fils et descendants). De plus, nous fournissons une traduction effective et en temps polynomial des expressions k-Downward XPath vers les dSTAs. De cette manière, nous pouvons réutiliser nos algorithmes conçus pour les dSTAs avec des expressions k-Downward XPath, et en particulier notre algorithme permettant d'évaluer au plus tôt.

Borner la concurrence et le délai Enfin, nous prouvons que pour les requêtes définies par dSTAs, il peut être décidé en temps polynomial si une requête a un délai borné et/ou une concurrence bornée. Le *délai* est le nombre maximal d'événements entre la lecture d'un nœud (ou d'un n-uplet de nœuds dans le cas n-aire) et le premier événement à partir duquel sa sélection peut être décidée. Le délai et la concurrence sont deux mesures clés pour la streamabilité : le délai est lié à la qualité de service, alors que la concurrence est une mesure de la quantité de mémoire nécessaire. Pour obtenir ces propriétés, nous utilisons et étendons les résultats concernant les relations reconnaissables d'arbres, déjà étudiées pour les arbres d'arité bornée [Tis90, CDG$^+$07] ainsi que les arbres d'arité non bornée [BL02, BLN07]. Ces relations entre arbres ont la particularité d'être reconnues

par des automates, modulo un codage des relations entre arbres vers les langages d'arbres. Nous montrons qu'il peut être décidé en temps polynomial si la valuation d'une relation reconnaissable binaire est bornée, et si elle est bornée par un certain k donné. Nous obtenons ces résultats par réduction sur la valuation bornée des transducteurs d'arbres [Sei92] et l'ambiguité k-bornée des automates d'arbres. Cela nous permet de décider en temps polynomial si, pour un k donné et une arité n donnée, une requête a un délai borné par k et/ou une concurrence bornée par k.

Index

Bibliography

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, 2004. (Cited page 12)

[ABD+05] Loredana Afanasiev, Patrick Blackburn, Ioanna Dimitriou, Bertrand Gaiffe, Evan Goris, Maarten Marx, and Maarten de Rijke. PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135, 2005. (Cited page 51)

[ABL07] Marcelo Arenas, Pablo Barceló, and Leonid Libkin. Combining temporal logics for querying XML documents. In *International Conference on Database Theory*, pages 359–373, 2007. (Cited page 52)

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 2000. (Cited page 52)

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *26th International Conference on Very Large Data Bases*, pages 53–64. Morgan Kaufmann, 2000. (Cited pages 5, 13, 123, and 192)

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. (Cited pages 49 and 50)

[Alu07] Rajeev Alur. Marrying words and trees. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 233–242. ACM-Press, 2007. (Cited pages 8, 32, 76, 79, 84, and 195)

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *36th ACM Symposium on Theory of Computing*, pages 202–211. ACM-Press, 2004. (Cited pages 8, 18, 32, 76, 80, and 195)

[AM06] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In *10th International Conference on Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, 2006. (Cited page 18)

[AM09] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009. (Cited pages 77 and 81)

[AMR08] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In *Developments in Language Theory, 12th International Conference*, volume 5257 of *Lecture Notes in Computer Science*, pages 108–120. Springer Verlag, 2008. (Cited pages 146, 151, 158, and 161)

[AU71] Alfred V. Aho and Jeffrey D. Ullmann. Translations on a context-free grammar. *Information and Control*, 19:439–475, 1971. (Cited page 32)

[Bag06] Guillaume Bagan. MSO Queries on Tree Decomposable Structures are Computable with Linear Delay. In *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer Verlag, 2006. (Cited pages 53 and 54)

[Bag09] Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques*. PhD thesis, Université de Caen, 2009. (Cited page 53)

[BBC02] Oliver Becker, Paul Brown, and Petr Cimprich. Streaming Transformations for XML (STX), 2002. `http://stx.sourceforge.net/`. (Cited pages 3, 18, and 190)

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16. ACM-Press, 2002. (Cited page 10)

[BC04] Mikołaj Bojańczyk and Thomas Colcombet. Tree-walking automata cannot be determinized. In *31st International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 246–256. Springer Verlag, 2004. (Cited page 32)

[BC05] Mikołaj Bojańczyk and Thomas Colcombet. Tree-walking automata do not recognize all regular languages. In *37th Annual ACM Symposium on Theory of Computing*, pages 234–243, New York, NY, USA, 2005. ACM-Press. (Cited page 32)

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003. (Cited pages 19 and 52)

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernàndez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language, W3C recommendation, 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/. (Cited pages 3, 43, and 189)

[BCG+03] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *19th International Conference on Data Engineering*, pages 455–466, 2003. (Cited pages 7, 15, and 193)

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer Verlag, 2007. (Cited page 50)

[BDGO08] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *RAIRO*, 42:147–164, 2008. (Cited page 53)

[BDM+06] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *Twenty-fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 10–19, 2006. (Cited page 47)

[Ber06] Alexandru Berlea. Online evaluation of regular tree queries. *Nordic Journal of Computing*, 13(4):1–26, 2006. (Cited pages 6, 9, 17, 76, 94, 193, and 196)

[Ber07] Alexandru Berlea. On-the-fly tuple selection for XQuery. In *Proceedings of the International Workshop on XQuery Implementation, Experience and Perspectives*, June 2007. (Cited pages 6, 17, and 193)

[BFG08] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2):1–79, 2008. (Cited pages 45 and 47)

[BFK05] Michael Benedikt, Wenfei Fan, and Gabriel Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005. (Cited page 47)

[BFLS06] François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Efficient evaluation of n-ary conjunctive queries over trees and graphs. In *8th annual ACM international workshop on Web Information and Data Management (WIDM)*, pages 11–18. ACM-Press, 2006. (Cited page 50)

[BGMM09] Henrik Björklund, Wouter Gelade, Marcel Marquardt, and Wim Martens. Incremental XPath evaluation. In *12th International Conference on Database Theory*, volume 361, pages 162–173. ACM-Press, 2009. (Cited pages 11 and 53)

[BJ07] Michael Benedikt and Alan Jeffrey. Efficient and expressive tree filters. In *Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 461–472. Springer Verlag, 2007. (Cited pages 7, 12, 14, 51, 58, 69, 95, 123, and 193)

[BJLW08] Michael Benedikt, Alan Jeffrey, and Ruy Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD International Conference on Management of Data*, pages 487–498. ACM-Press, 2008. (Cited pages 6, 7, 9, 12, 14, 94, 123, 186, 193, 194, and 195)

[BK93] Anne Brüggemann-Klein. Regular expressions to finite automata. *Theoretical Computer Science*, 120(2):197–213, November 1993. (Cited page 82)

[BK08] Michael Benedikt and Christoph Koch. XPath leashed. *ACM computing surveys*, 41(1), 2008. (Cited pages 43 and 45)

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD'02*, pages 310–321, 2002. (Cited pages 47 and 55)

[BKW98] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998. (Cited page 35)

[BKW00] Anne Brüggemann-Klein and Derick Wood. Caterpillars, context, tree automata and tree pattern matching. In *Developments in Language Theory, Foundations, Applications, and Perspectives (1999)*, pages 270–285. World Scientific, 2000. (Cited page 52)

[BKWM01] Anne Brüggemann-Klein, Derick Wood, and Makoto Murata. Regular tree and regular hedge languages over unranked alphabets: Version 1, April 07 2001. (Cited page 32)

[BL02] Michael Benedikt and Leonid Libkin. Tree extension algebras: Logics, automata, and query languages. In *Proceeding of the 9^{th} Logic in Computer Science Conference*, pages 203–214. IEEE Comp. Soc. Press, 2002. (Cited pages 10, 146, 147, and 196)

[BL05] Pablo Barceló and Leonid Libkin. Temporal logics over unranked trees. In *20th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40. IEEE Comp. Soc. Press, 2005. (Cited pages 51 and 52)

[BLN07] Michael Benedikt, Leonid Libkin, and Frank Neven. Logical definability and query languages over ranked and unranked trees. *ACM Transactions on Computational Logics*, 8(2), April 2007. (Cited pages 10, 146, 147, 162, 164, 165, and 196)

[BLS06] Vince Bárány, Christof Löding, and Olivier Serre. Regularity problems for visibly pushdown languages. In B. Durand and W. Thomas, editors, *23rd Annual Symposioum on Theoretical Aspects of Computer Science*, volume 3884 of *Lecture Notes in Computer Science*, pages 420–431. Springer Verlag, 2006. (Cited page 13)

[Boj04] Mikołaj Bojańczyk. *Decidable Properties of Tree Languages*. PhD thesis, Warsaw University, 2004. (Cited pages 28 and 29)

[Boj08] Mikołaj Bojańczyk. Effective characterizations of tree logics, 2008. PODS'08 Keynote. (Cited page 26)

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. (Cited pages 1, 3, 33, 35, 43, 187, and 189)

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. (Cited page 123)

[BS04] Alexandru Berlea and Helmut Seidl. Binary queries for document trees. *Nordic Journal of Computing*, 11(1):41–71, 2004. (Cited pages 4, 17, 52, 77, 88, and 190)

[BS05] Michael Benedikt and Luc Segoufin. Regular tree languages definable in FO and FOmod. In *22nd International Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *Lecture Notes in Computer Science*, pages 327–339. Springer Verlag, 2005. (Cited page 28)

[BSSS06] Mikołaj Bojańczyk, Mathias Samuelides, Thomas Schwentick, and Luc Segoufin. Expressive power of pebbles automata. In *International Colloquium on Automata Languages and Programming (ICALP'06)*, Lecture Notes in Computer Science, pages 157–168. Springer Verlag, 2006. (Cited page 32)

[Büc60] J.R. Büchi. On a decision method in a restricted second order arithmetic. In Stanford Univ. Press., editor, *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, pages 1–11, 1960. (Cited page 29)

[BYFJ04] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of XPath evaluation over XML streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 177–188. ACM-Press, 2004. (Cited pages 7, 11, 14, and 194)

[BYFJ05] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over XML streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 216–227. ACM-Press, 2005. (Cited pages 4, 5, 7, 9, 12, 15, 57, 61, 62, 66, 68, 74, 94, 185, 191, 192, 193, and 195)

[BYFJ07] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of XPath evaluation over XML streams. *Journal of Computer and System Science*, 73(3):391–441, 2007. (Cited pages 7, 11, 14, 193, and 194)

[CD99] James Clark and Steve DeRose. XML path language (XPath): W3C recommendation, 1999. (Cited pages 3, 43, and 189)

[CDG+07] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available online since 1997: `http://tata.gforge.inria.fr`, October 2007. Revised October, 12th 2007. (Cited pages 4, 10, 16, 23, 31, 32, 33, 42, 75, 146, 162, 163, 164, 165, 190, and 196)

[CDGLV09] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Vardi. An Automata-Theoretic Approach to Regular XPath. In *12th International Symposium on Database Programming Languages*, 2009. (Cited page 124)

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An Efficient XPath Query Processor for XML Streams. In *22nd International Conference on Data Engineering*, page 79. IEEE Computer Society, 2006. (Cited page 15)

[CFGR02] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002. (Cited page 14)

[CG04] Luca Cardelli and Giorgio Ghelli. TQL: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004. (Cited page 52)

[CGLN09] Jérôme Champavère, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Efficient inclusion checking for deterministic tree automata and XML schemas. *Information and Computation*, 2009. (Cited pages 33, 37, and 83)

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Know. Data Eng.*, 1(1):146–166, March 1989. (Cited page 51)

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Verlag, 1990. (Cited page 50)

[Che06] Dunren Che. MyTwigStack: A Holistic Twig Join Algorithm with Effective Path Merging Support. In *7th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 184–189. IEEE Computer Society, 2006. (Cited page 55)

[Chi00] Boris Chidlovskii. Using regular tree automata as XML schemas. In *Proceedings of IEEE Advances in Digital Libraries*, pages 89–98, 2000. (Cited page 33)

[Chu36] Alonzo Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. (Cited page 28)

[CJdR08] Huang Cheng, Li Jun, and Michel de Rougemont. Approximate Validity of XML Streaming Data. In *9th International Conference on Web-Age Information Management (WAIM)*, pages 149–156, 2008. (Cited page 186)

[Cla99] James Clark. XSL Transformations (XSLT) version 1.0, W3C recommendation, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116. (Cited pages 3, 43, and 189)

[Cla08] Robert Clark. Querying Streaming XML Using Visibly Pushdown Automata. Technical Report UIUCDCS-R-2008-3008, University of Illinois at Urbana-Champaign, October 2008. (Cited pages 16 and 124)

[CLN04] Julien Carme, Aurélien Lemay, and Joachim Niehren. Learning node selecting tree transducer from completely annotated examples. In *7th International Colloquium on Grammatical Inference*, volume 3264 of *Lecture Notes in Artificial Intelligence*, pages 91–102. Springer Verlag, 2004. (Cited page 152)

[CLT+06] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of VLDB*, pages 283–294, 2006. (Cited page 55)

[CLT+08] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Scalable Filtering

of Multiple Generalized-Tree-Pattern Queries over XML Streams. *IEEE Trans. on Know. Data Eng.*, 20(12):1627–1640, 2008. (Cited page 15)

[CM01] James Clark and Makoto Murata. Relax NG specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001. (Cited page 33)

[CNT04] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In *19th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer Verlag, 2004. (Cited pages 4, 31, 77, 88, and 190)

[Cou09] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009. (Cited page 53)

[CR04] Cristiana Chitic and Daniela Rosu. On validation of XML streams using finite state machines. In *7th International Workshop on the Web and Databases (WebDB)*, pages 85–90. ACM-Press, 2004. (Cited page 13)

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM computing surveys*, 33(3):374–425, September 2001. (Cited page 51)

[Des01] Arpan Desai. Introduction to Sequential XPath. In *IDEAlliance XML Conference*, 2001. (Cited page 16)

[DFFT02] Y. Diao, P. Fischer, M.J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *18th International Conference on Data Engineering*, page 341. IEEE Comp. Soc. Press, 2002. (Cited page 14)

[DMJ01] Steve DeRose, Eve Maler, and Ron Daniel Jr. XML Pointer Language (XPointer) version 1.0, 2001. http://www.w3.org/TR/2001/WD-xptr-20010108/. (Cited pages 3, 43, and 189)

[DO06] Arnaud Durand and Frédéric Olive. First-Order Queries over One Unary Function. In *CSL, 15th Annual Conference of the EACSL*,

volume 4207 of *Lecture Notes in Computer Science*, pages 334–348. Springer Verlag, 2006. (Cited page 54)

[Don65] John E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965. (Cited page 30)

[Don70] John E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Science*, 4:406–451, 1970. (Cited pages 11, 16, 23, 29, 30, 32, and 124)

[DR07] Jana Dvoráková and Branislav Rovan. A Transducer-Based Framework for Streaming XML Transformations. In *33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 50–60. Institute of Computer Science AS CR, Prague, 2007. (Cited page 18)

[DRF04] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale XML dissemination service. In *30th international conference on Very Large Data Bases*, pages 612–623. VLDB Endowment, 2004. (Cited page 14)

[dRV08] Michel de Rougemont and Adrien Vieilleribière. Approximate schemas, source-consistency and query answering. *Journal of Intelligent Information Systems*, 31(2):127–146, 2008. (Cited page 186)

[EF99] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory*. Springer Verlag, Berlin, 1999. (Cited pages 26 and 28)

[EH99] Joost Engelfriet and Hendrik Jan Hoogeboom. Tree-walking pebble automata. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999. (Cited page 32)

[EH07] Joost Engelfriet and Hendrik Jan Hoogeboom. Automata with nested pebbles capture first-order logic with transitive closure. *Logical Methods in Computer Science*, 3(2):3, 2007. (Cited page 32)

[EHB99] Joost Engelfriet, Hendrik Jan Hoogeboom, and Jan-Pascal Van Best. Trips on trees. *Acta Cybern.*, 14(1):51–64, 1999. (Cited page 32)

[Fag75] Ronald Fagin. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:89–96, 1975. (Cited page 28)

[FG02] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 215–224, Washington, DC, USA, 2002. (Cited page 30)

[FGK03] Markus Frick, Martin Grohe, and Christoph Koch. Query Evaluation on Compressed Trees. In *18th IEEE Symposium on Logic in Computer Science*, pages 188–197. IEEE Comp. Soc. Press, 2003. (Cited pages 25, 42, and 152)

[FHM+05] Mary F. Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In *16th International Conference on Database and Expert Systems Applications (DEXA)*, pages 554–563, 2005. (Cited pages 5 and 192)

[Fig09] Diego Figueira. Satisfiability of downward XPath with data equality tests. In *28th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 197–206. ACM-Press, 2009. (Cited page 47)

[FMSS07] Mary Fernandez, Philippe Michiels, Jérôme Siméon, and Michael Stark. XQuery Streaming à la Carte. In *23nd International Conference on Data Engineering*, pages 256–265. IEEE Comp. Soc. Press, 2007. (Cited page 18)

[FN07] Alain Frisch and Keisuke Nakano. Streaming XML transformation using term rewriting. In *Programming Language Technologies for XML (PLAN-X 2007)*, 2007. (Cited page 19)

[FNTT06] Emmanuel Filiot, Joachim Niehren, Jean-Marc Talbot, and Sophie Tison. Composing monadic queries in trees. In Giuseppe Castagna and Mukund Raghavachari, editors, *PLAN-X International Workshop*, pages 61–70. Basic Research in Computer Science, 2006. (Cited page 52)

[FNTT07] Emmanuel Filiot, Joachim Niehren, Jean-Marc Talbot, and Sophie Tison. Polynomial time fragments of XPath with variables. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 205–214. ACM-Press, 2007. (Cited pages 49 and 54)

[Fri04] Alain Frisch. Regular tree language recognition with static information. In *Exploring New Frontiers of Theoretical Informatics, IFIP*

18th World Computer Congress, TCS 3rd International Conference on Theoretical Computer Science, pages 661–674, 2004. (Cited pages 19 and 76)

[FTT07] Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Satisfiability of a spatial logic with tree variables. In *16th EACSL Annual Conference on Computer Science and Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 130–145. Springer Verlag, 2007. (Cited page 52)

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`. (Cited pages 3, 33, 43, and 189)

[GC07a] Gang Gou and Rada Chirkova. Efficient algorithms for evaluating XPath over streams. In *36th ACM SIGMOD International Conference on Management of Data*, pages 269–280. ACM-Press, 2007. (Cited pages 7, 15, 94, and 193)

[GC07b] Gang Gou and Rada Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Trans. on Know. Data Eng.*, 19(10):1381–1403, 2007. (Cited page 55)

[GF05] Floris Geerts and Wenfei Fan. Satisfiability of XPath Queries with Sibling Axes. In *10th International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 122–137. Springer Verlag, 2005. (Cited page 47)

[GGM+04] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004. (Cited pages 14 and 123)

[GGV02] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a Deductive Query Language with Linear Time Model Checking. *ACM Transactions on Computational Logics*, 3(1):42–79, January 2002. (Cited page 51)

[GK04] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *Journal of the ACM*, 51(1):74–113, 2004. (Cited page 50)

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath processing in a nutshell. *SIGMOD Rec.*, 32(2):21–27, 2003. (Cited pages 45 and 54)

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005. (Cited pages 3, 43, and 54)

[GKPS05] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the ACM*, 52(2):284–335, 2005. (Cited pages 13 and 54)

[GKS06] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *Journal of the ACM*, 53(2):238–272, 2006. (Cited page 50)

[GKS07] Martin Grohe, Christoph Koch, and Nicole Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *Theoretical Computer Science*, 380(1-2):199–217, 2007. (Cited pages 7, 10, 11, 12, 58, 185, and 194)

[GLdB07] Yuri Gurevich, Dirk Leinders, and Jan Van den Bussche. A Theory of Stream Queries. In *Database Programming Languages*, Lecture Notes in Computer Science, pages 153–168. Springer Verlag, 2007. (Cited page 10)

[GLS07] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, volume 42, pages 342–351. ACM-Press, 2007. (Cited pages 45 and 52)

[Glu61] Victor M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961. (Cited page 35)

[GM05] Evan Goris and Maarten Marx. Looping caterpillars. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 51–60. IEEE Comp. Soc. Press, June 2005. (Cited page 52)

[GMOS03] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT'03*, volume 2572 of *Lecture Notes in Computer Science*, pages 173–189. Springer Verlag, 2003. (Cited pages 14 and 123)

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD International Conference on Management of Data*, pages 157–166. ACM Press, 1993. (Cited pages 11 and 53)

[GO04] Etienne Grandjean and Frédéric Olive. Graph properties checkable in linear time in the number of vertices. *Journal of Computer and System Science*, 68(3):546–597, 2004. (Cited page 62)

[GP09] Olivier Gauwin and Benoît Papegay. EvoXs project, 2009. `https://gforge.inria.fr/projects/evoxs/`. (Cited page 115)

[Gra96] Etienne Grandjean. Sorting, Linear Time and the Satisfiability Problem. *Ann. Math. Artif. Intell.*, 16:183–236, 1996. (Cited page 62)

[Grz53] Andrzej Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–45, 1953. (Cited page 30)

[GS03a] Etienne Grandjean and Thomas Schwentick. Machine-Independent Characterizations and Complete Problems for Deterministic Linear Time. *SIAM Journal on Computing*, 32(1):196–230, 2003. (Cited page 53)

[GS03b] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM-Press, 2003. (Cited pages 14 and 123)

[Hid03] Jan Hidders. Satisfiability of XPath expressions. In *The 9th International Workshop on Data Base Programming Languages*, pages 21–36, 2003. (Cited page 49)

[HJHL08] Wook-Shin Han, Haifeng Jiang, Howard Ho, and Quanzhong Li. StreamTX: extracting tuples from streaming XML data. *Proceedings of the VLDB Endowment*, 1(1):289–300, 2008. (Cited page 15)

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1), 1985. (Cited page 51)

[HP03] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 6(13):961–1004, 2003. (Cited page 43)

[HRR99] Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. *Computing on data streams*. American Mathematical Society, 1999. (Cited page 12)

[HT87] Thilo Hafer and Wolfgang Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *14th International Colloquium on Automata, languages and programming*, pages 269–279. Springer Verlag, 1987. (Cited page 51)

[HU69] J.E. Hopcroft and J.D. Ullman. Some Results on Tape-bounded Turing Machines. *Journal of the ACM*, 16(1):168–177, 1969. (Cited page 10)

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. (Cited pages 3 and 190)

[IHW02] Zachary G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002. (Cited page 18)

[Jel06] Rick Jelliffe. Schematron specification (ISO/IEC 19757-3), 2006. (Cited page 33)

[JFB05] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005. (Cited page 15)

[JLH+07] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, Qiang Zhu, and Dunren Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *18th International Conference on Database and Expert Systems Applications (DEXA)*, Lecture Notes in Computer Science, pages 87–97. Springer Verlag, 2007. (Cited page 55)

[JPY88] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988. (Cited page 53)

[JWLY03] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic Twig Joins on Indexed XML Documents. In *29th International Conference on Very Large Data Bases*, pages 273–284, 2003. (Cited page 55)

[Kam68] J. A. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968. (Cited page 51)

[Kay09] Michael Kay. Saxon diaries, June 19th, 2009. `http://saxonica.blogharbor.com/blog`. (Cited pages 4 and 190)

[KMV07] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly Pushdown Automata for Streaming XML. In *16th international conference on World Wide Web*, pages 1053–1062. ACM-Press, 2007. (Cited pages 13, 17, 76, 82, and 124)

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997. (Cited page 11)

[Koc03] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *Proc. VLDB 2003*, 2003. (Cited pages 4, 25, and 190)

[Koc06] Christoph Koch. Processing queries on tree-structured data efficiently. In *25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 213–224, New York, NY, USA, 2006. ACM-Press. (Cited page 52)

[KRS+07] Michael Kay, Jonathan Robie, Jérôme Siméon, Scott Boag, Mary F. Fernández, Anders Berglund, and Don Chamberlin. XML path language (XPath) 2.0. Recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/. (Cited page 48)

[KS07] Christoph Koch and Stefanie Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLPB Journal*, 16(3):317–342, 2007. (Cited page 18)

[KSSS04a] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *30th International Conference on Very Large Data Bases*, pages 1309–1312. Morgan Kaufmann, 2004. (Cited page 18)

[KSSS04b] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *30th International Conference on Very Large Data Bases*, pages 228–239. Morgan Kaufmann, 2004. (Cited page 18)

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001. (Cited page 95)

[LB09] Clemens Ley and Michael Benedikt. How Big Must Complete XML Query Languages Be? In *12th International Conference on Database Theory*, pages 183–200. ACM-Press, 2009. (Cited pages 7, 11, 16, 185, and 194)

[Lib04] Leonid Libkin. *Elements of Finite Model Theory*. Springer Verlag, 2004. (Cited pages 26 and 28)

[Lib06] Leonid Libkin. Logics over unranked trees: an overview. *Logical Methods in Computer Science*, 3(2):1–31, 2006. (Cited pages 24, 26, 51, and 123)

[LMP02] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based XML query processor. In *28th international conference on Very Large Data Bases*, pages 227–238. VLDB Endowment, 2002. (Cited page 18)

[LS08] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with Temporal Logics and Automata. In *15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 97–112. Springer Verlag, 2008. (Cited pages 121 and 124)

[Mar04a] Maarten Marx. Conditional XPath, the first order complete XPath dialect. In *ACP SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 13–22. ACM-Press, 2004. (Cited page 46)

[Mar04b] Maarten Marx. XPath with conditional axis relations. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *9th International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*,

pages 477–494. Springer Verlag, 2004. (Cited pages 46, 47, 52, and 54)

[Mar05a] Maarten Marx. Conditional XPath. *ACM Transactions on Database Systems*, 30(4):929–959, 2005. (Cited pages 45, 46, 51, and 124)

[Mar05b] Maarten Marx. First order paths in ordered trees. In *International Conference on Database Theory*, pages 114–128, 2005. (Cited page 45)

[MdR05] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Rec.*, 34(2):41–46, 2005. (Cited pages 11, 45, and 124)

[Mey73] Albert R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. Manuscript, 1973. (Cited page 30)

[Mic07] Philippe Michiels. *Optimizing XPath in the Context of an XQuery Implementation*. PhD thesis, Universiteit Antwerpen, 2007. (Cited pages 5 and 192)

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001. (Cited pages 4, 33, 35, 36, 37, and 190)

[MNS04] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science 2004, 29th International Symposium*, pages 889–900, 2004. (Cited page 35)

[MNS05] Wim Martens, Frank Neven, and Thomas Schwentick. Which XML schemas admit 1-pass preorder typing? In *10th International Conference on Database Theory*, volume 3363 of *Lecture Notes in Computer Science*, pages 68–82. Springer Verlag, 2005. (Cited pages 13, 36, and 37)

[MNSB06a] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML schema. *ACM Transactions of Database Systems*, 31(3):770–813, 2006. (Cited page 13)

[MNSB06b] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and Complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, September 2006. (Cited pages 33, 36, and 55)

[Mor94] Etsuro Moriya. On two-way tree automata. *Information Processing Letters*, 50(3):117–121, 1994. (Cited page 43)

[MS03] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *29th International Conference on Very Large Data Bases*, pages 213–224. Morgan Kaufmann, 2003. (Cited page 54)

[MS04] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004. (Cited pages 45, 46, and 47)

[MSB01] Holger Meuss, Klaus U. Schulz, and François Bry. Towards aggregated answers for semistructured data. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 346–360. Springer Verlag, 2001. (Cited pages 65 and 186)

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Type checking XML transformers. *Journal of Computer and System Science*, 1(66):66–97, 2003. (Cited page 25)

[Mut05] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005. (Cited page 10)

[MV08] Parthasarathy Madhusudan and Mahesh Viswanathan. Query Automata for Nested Words, 2008. Unpublished. (Cited pages 18 and 95)

[Nak04] Keisuke Nakano. An Implementation Scheme for XML Transformation Languages through Derivation of Stream Processors. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS04)*, 2004. (Cited page 19)

[Neu00] Andreas Neumann. *Parsing and Quering XML Documents in SML*. PhD thesis, Universität Trier, 2000. (Cited page 17)

[Nev02a] Frank Neven. Automata, logic, and XML. In *Computer Science Logic*, Lecture Notes in Computer Science, pages 2–26. Springer Verlag, 2002. (Cited pages 24, 30, and 42)

[Nev02b] Frank Neven. Automata theory for XML researchers. *SIGMOD Rec.*, 31(3):39–46, 2002. (Cited pages 24, 30, and 42)

[NK08] Abdul Nizar and Sreenivasa Kumar. Efficient Evaluation of Forward XPath Axes over XML Streams. In *14th International Conference on Management of Data (COMAD)*, pages 222–233, 2008. (Cited pages 7, 16, and 193)

[NK09] Abdul Nizar and Sreenivasa Kumar. Ordered Backward XPath Axis Processing against XML Streams. In *6th International XML Database Symposium (XSym)*, volume 5679 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2009. (Cited pages 7, 16, and 193)

[NPTT05] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary queries by tree automata. In *10th International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 217–231. Springer Verlag, September 2005. (Cited pages 42 and 152)

[NS98] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *18th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145. Springer Verlag, 1998. (Cited pages 8, 17, 32, 52, 76, 85, and 195)

[NS00] Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-00)*, pages 145–156, N. Y., May 15–17 2000. (Cited page 52)

[NS02] Frank Neven and Thomas Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002. (Cited pages 4, 43, 124, and 190)

[NS03] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *9th International Conference on Database Theory*, volume 2572 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, 2003. (Cited page 45)

[OKB03] Dan Olteanu, Tobias Kiesling, and François Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *19th International Conference on Data Engineering*, pages 702–704, 2003. (Cited pages 7, 16, and 193)

[Olt07a] Dan Olteanu. Forward node-selecting queries over trees. *ACM Transactions on Database Systems*, 32(1):3, 2007. (Cited page 16)

[Olt07b] Dan Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007. (Cited pages 7, 14, 16, 46, 95, 123, and 193)

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer Verlag, 2002. (Cited pages 7, 16, 46, and 193)

[Par09] Paweł Parys. XPath Evaluation in Linear Time with Polynomial Combined Complexity. In *28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 55–64. ACM-Press, 2009. (Cited page 47)

[PC05] Feng Peng and Sudarshan S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems*, 30(2):577–623, 2005. (Cited pages 5, 15, 123, and 191)

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *19th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 35–46. ACM-Press, 2000. (Cited pages 35 and 36)

[Rab69] Michael O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969. (Cited page 25)

[Ram03] Prakash Ramanan. Covering indexes for XML queries: bisimulation - simulation = negation. In *29th international conference on Very Large Data Bases*, pages 165–176. Morgan Kaufmann, 2003. (Cited page 54)

[Ram05] Prakash Ramanan. Evaluating an XPath Query on a Streaming XML Document. In *12th International Conference on Management of Data (COMAD)*, 2005. (Cited pages 7, 15, and 193)

[Ram09] Prakash Ramanan. Worst-case optimal algorithm for XPath evaluation over XML streams. *Journal of Computer and System Science*, 2009. Article in press. (Cited pages 7, 15, and 193)

[RNC03] George Russell, Mathias Neumüller, and Richard C. H. Connor. TypEx: A Type Based Approach to XML Stream Querying. In *International Workshop on Web and Databases (WebDB)*, pages 55–60, 2003. (Cited page 13)

[SBY08] Mirit Shalem and Ziv Bar-Yossef. The Space Complexity of Processing XML Twig Queries Over Indexed Documents. In *24th International Conference on Data Engineering*, pages 824–832, 2008. (Cited page 186)

[Sch92] Bernd-Holger Schlingloff. Expressive Completeness Of Temporal Logic Of Trees. *Journal of Applied Non-Classical Logics*, 2(2):157–180, 1992. (Cited page 123)

[Sch00] Thomas Schwentick. On diving in trees. In *MFCS '00: Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, pages 660–669, London, UK, 2000. (Cited page 52)

[Sch07a] Nicole Schweikardt. Machine models and lower bounds for query processing. In *Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 41–52. ACM-Press, 2007. (Cited pages 10 and 58)

[Sch07b] Thomas Schwentick. Automata for XML—a survey. *Journal of Computer and System Science*, 73(3):289–315, 2007. (Cited pages 24, 30, 32, and 33)

[Sch09] Nicole Schweikardt. Lower Bounds for Multi-Pass Processing of Multiple Data Streams. In *26th International Symposium on Theoretical Aspects of Computer Science*, volume 09001, pages 51–61, 2009. Dagstuhl Seminar Proceedings. (Cited page 12)

[SdS08] Jacques Sakarovitch and Rodrigo de Souza. On the Decidability of Bounded Valuedness for Transducers. In *33rd international symposium on Mathematical Foundations of Computer Science*, volume 5162 of *Lecture Notes in Computer Science*, pages 588–600. Springer Verlag, 2008. (Cited page 172)

[Seg03] Luc Segoufin. Typing and querying XML documents: some complexity bounds. In *22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 167–178, 2003. (Cited page 55)

[Sei92] Helmut Seidl. Ambiguity, Valuedness and Costs, 1992. Habilitation Thesis. Universität des Saarlandes. (Cited pages 10, 147, 158, 161, 162, 164, 169, 170, 172, 173, and 197)

[SH85] R. E. Stearns and H. B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985. (Cited pages 146 and 151)

[SI84] Oded Shmueli and Alon Itai. Maintenance of Views. In *SIGMOD'84, Proceedings of Annual Meeting*, pages 240–255. ACM Press, 1984. (Cited pages 11 and 53)

[SK05] Stefanie Scherzinger and Alfons Kemper. Syntax-directed Transformations of XML Streams (TransformX). In *Programming Language Technologies for XML (PLAN-X 2005)*, 2005. (Cited page 19)

[SM73] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *Proc. 5th ACM Symp. on Theory of Computing*, pages 1–9, 1973. (Cited page 30)

[SS07] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *Database Theory - ICDT 2007, 11th International Conference*, pages 299–313, 2007. (Cited page 12)

[SSK07] Michael Schmidt, Stefanie Scherzinger, and Christoph Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *23rd IEEE International Conference on Data Engineering*, pages 236–245, 2007. (Cited page 18)

[Sto74] L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory*. PhD thesis, Department of Electrical Engineering, MIT, 1974. (Cited pages 28 and 30)

[STW08] Klaus Dieter Schewe, Bernhard Thalheim, and Qing Wang. Validation of streaming XML documents with abstract state machines. In *10th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 147–153. ACM-Press, 2008. (Cited pages 13 and 186)

[SV02] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 53–64, 2002. (Cited pages 5, 12, 13, 35, 58, and 191)

[Tak75] Masako Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27:1–36, 1975. (Cited page 17)

[tC06] Balder ten Cate. The expressiveness of XPath with transitive closure. In *25th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 328–337. ACM-Press, 2006. (Cited page 46)

[tCL07] Balder ten Cate and Carsten Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *26th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 73–82. ACM-Press, 2007. (Cited page 49)

[tCM07] Balder ten Cate and Maarten Marx. Axiomatizing the logical core of XPath 2.0. In *International Conference on Database Theory*, volume 4353 of *Lecture Notes in Computer Science*, pages 134–148. Springer Verlag, 2007. (Cited pages 48 and 49)

[tCS08] Balder ten Cate and Luc Segoufin. XPath, transitive closure logic, and nested tree walking automata. In *27th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 251–260. ACM-Press, 2008. (Cited pages 32 and 47)

[Tho97] Wolfgang Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer Verlag, 1997. (Cited pages 23 and 42)

[Tis90] Sophie Tison. *Automates Comme Outils de Décision dans les Arbres*. Thèse d'habilitation, Laboratoire d'Informatique Fondamentale de Lille, 1990. (Cited pages 10, 146, and 196)

[Tra50] B.A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950. (Cited page 28)

[Tur37] Alan M. Turing. Computability and lambda-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937. (Cited page 28)

[TVY08] Alex Thomo, S. Venkatesh, and Ying Ying Ye. Visibly Pushdown Transducers for Approximate Validation of Streaming XML. In *5th International Symposium on Foundations of Information and Knowledge Systems (FoIKS)*, volume 4932 of *Lecture Notes in*

Computer Science, pages 219–238. Springer Verlag, 2008. (Cited page 186)

[TW65] J. W. Thatcher and J. B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 820, 1965. Abstract No 65T-649. (Cited page 30)

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968. (Cited pages 11, 16, 23, 29, 30, 32, and 124)

[Var82] Moshe Y. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–146, 1982. (Cited pages 28 and 30)

[Var95] Moshe Y. Vardi. On the complexity of bounded-variable queries. In *Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 266–276, 1995. (Cited page 53)

[vdV03] Eric van der Vlist. *Relax NG*. O'Reilly Media, Inc., 2003. (Cited pages 3 and 189)

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM computing surveys*, 33(2):209–271, 2001. (Cited page 10)

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994. (Cited pages 121 and 124)

[WMT09] Norman Walsh, Alex Milowski, and Henry S. Thompson. XProc: An XML Pipeline Language, March 2009. W3C Candidate Recommendation. (Cited pages 3 and 189)

[Woo03] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *9th International Conference on Database Theory*, volume 2572 of *Lecture Notes in Computer Science*, pages 300–314. Springer Verlag, 2003. (Cited page 45)

[WRML08] Mingzhu Wei, Elke A. Rundensteiner, Murali Mani, and Ming Li. Processing recursive XQuery over XML streams: The Raindrop approach. *Data Knowl. Eng.*, 65(2):243–265, 2008. (Cited page 18)

[WS86] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. In *Mathematical Foundations of Computer Science*, volume 233 of *Lecture Notes in Computer Science*, pages 620–629. Springer Verlag, 1986. (Cited pages 146 and 151)

[WT08] Xiaoying Wu and Dimitri Theodoratos. Evaluating Partial Tree-Pattern Queries on XML Streams. In *17th ACM International Conference on Information and Knowledge Management (CIKM'08)*, pages 1409–1410. ACM Press, 2008. (Cited pages 7, 15, and 193)

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceeding of VLDB*, pages 82–94. IEEE Computer Society, 1981. (Cited page 50)

[Yao79] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *11th annual ACM Symposium on Theory of Computing*, pages 209–213. ACM-Press, 1979. (Cited page 11)

[ZXM07] Junfeng Zhou, Min Xie, and Xiaofeng Meng. TwigStack+: Holistic twig join pruning using extended solution extension. *Wuhan University Journal of Natural Sciences*, 12(5):855–860, 2007. (Cited page 55)

Notations

Notation	Description	Section
	Generic Notations	
$\mathbb{N}$	set of strictly positive natural numbers	2.1.1
$\mathbb{N}_0$	set of natural numbers including 0	2.1.1
$\flat$	Boolean	6.2.4
$\mathbb{B}$	the set of Booleans	7.5.1
$[\![.]\!]$	semantic interpretation	2.3.3
	Logic and Relational Structures	
Δ	relational signature	2.1.2
s	relational structure	2.1.2
$\mathcal{S}$	set of relational structures	2.1.2
μ	assignment of variables	2.1.2
$\mathcal{V}$	set of variables	2.3.3
$\mathcal{V}_n$	the set of variables $\{x_1, \ldots, x_n\}$	2.3.1
	Words, Trees, Terms and Hedges	
Σ	alphabet	2.1.1
a	element of Σ	2.1.1
w	word	2.1.2
t	tree	2.1.1
$\mathcal{T}_\Sigma$	unranked trees over Σ	2.1.1
π	node of a tree	2.1.1
τ	tuple of nodes of a tree	2.3.1
d	depth of a tree	2.1.1
h	hedge	2.1.1
$\mathcal{H}_\Sigma$	hedges over Σ	2.1.1

Notation	Description	Section
T	term	
	Queries	
Q	query	2.3.1
n	arity of a query	2.3.1
S	schema	2.3.1
L_Q	canonical language of the query Q	2.3.1
dom(Q)	domain of a query, i.e., associated schema	2.3.1
e	expression defining a query	2.3.1
$\mathcal{Q}$	query class	2.3.1
Q_A	query with universal schema, defined from the automaton A by $L_Q = L(A)$	2.3.2
$Q_{A,B}$	query Q_A, except that *dom*$(Q) = L(B)$	2.3.2
	Automata	
A	automaton	
$L(A)$	language recognized by the automaton A	
B	automaton recognizing the schema language	
q	a state	4.2
P	a set of states	
γ	a node state	4.2
stat	set of states	
stat$_{\text{e}}$	set of event states	4.2
stat$_{\text{n}}$	set of node states	4.2
init	set of initial states	
fin	set of final states	
rul	set of rules	
r	run of an automaton	
runs	set of runs of an automaton	
runs_succ	successful runs of an automaton	
amb(A)	degree of ambiguity of an automaton	7.3.1
	Streaming	
$\widehat{S}$	$\{op, cl\} \times S$	3.2.1
η	event	3.2

Notation	Description	Section
$\preceq$	document order on events	3.2.1
$pr(\eta)$	event preceding η in document order	3.2.1
α	action in $\{op, cl\}$	3.2
$\mathcal{M}$	Streaming Random Access Machine (SRAM)	3.2.4
m	degree of streamability	3.3.1
	Relations over Trees	
R	relation over trees	7.4.2
$\circledast$	overlay operator	7.4.2
$\boxdot$	fill symbol for differing structures	7.4.2
$ovl(R)$	language of overlays of the relation R	7.4.2
$\Re$	set of symbols of recognizable relations	7.4.4
r	symbol of recognizable relation	7.4.4
Ω	set of alphabets	7.4.4
ω	alphabet in Ω	7.4.4
Sorts	a set of sorts	7.4.3
σ	a sort	7.4.3
$sort(r)$	the sort of symbol r of recognizable relation	7.4.3
$val(R)$	valuedness of the relation R	7.4.5

List of Figures

Appendix A

Licence Creative Commons

Ce chapitre contient le texte de la licence Creative Commons Paternité – Pas d'Utilisation Commerciale – Pas de Modification, version 2.0.[1]

A.1 Contrat

L'Oeuvre (telle que définie ci-dessous) est mise à disposition selon les termes du présent contrat appelé Contrat Public Creative Commons (dénommé ici "CPCC" ou "Contrat"). L'Oeuvre est protégée par le droit de la propriété littéraire et artistique (droit d'auteur, droits voisins, droits des producteurs de bases de données) ou toute autre loi applicable. Toute utilisation de l'Oeuvre autrement qu'explicitement autorisée selon ce Contrat ou le droit applicable est interdite.

L'exercice sur l'Oeuvre de tout droit proposé par le présent contrat vaut acceptation de celui-ci. Selon les termes et les obligations du présent contrat, la partie Offrante propose à la partie Acceptante l'exercice de certains droits présentés ci-après, et l'Acceptant en approuve les termes et conditions d'utilisation.

Définitions

1. "Oeuvre" : oeuvre de l'esprit protégeable par le droit de la propriété littéraire et artistique ou toute loi applicable et qui est mise à disposition selon les termes du présent Contrat.

2. "Oeuvre dite Collective" : une oeuvre dans laquelle l'oeuvre, dans sa forme intégrale et non modifiée, est assemblée en un ensemble collectif avec d'autres contributions qui constituent en elles-mêmes des oeuvres séparées et indépendantes. Constituent notamment des Oeuvres dites Collectives les

[1] Voir : `http://creativecommons.org/licenses/by-nc-nd/2.0/fr/`.

publications périodiques, les anthologies ou les encyclopédies. Aux termes de la présente autorisation, une oeuvre qui constitue une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée (telle que définie ci-après).

3. “Oeuvre dite Dérivée” : une oeuvre créée soit à partir de l’Oeuvre seule, soit à partir de l’Oeuvre et d’autres oeuvres préexistantes. Constituent notamment des Oeuvres dites Dérivées les traductions, les arrangements musicaux, les adaptations théâtrales, littéraires ou cinématographiques, les enregistrements sonores, les reproductions par un art ou un procédé quelconque, les résumés, ou toute autre forme sous laquelle l’Oeuvre puisse être remaniée, modifiée, transformée ou adaptée, à l’exception d’une oeuvre qui constitue une Oeuvre dite Collective. Une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée aux termes du présent Contrat. Dans le cas où l’Oeuvre serait une composition musicale ou un enregistrement sonore, la synchronisation de l’oeuvre avec une image animée sera considérée comme une Oeuvre dite Dérivée pour les propos de ce Contrat.

4. “Auteur original” : la ou les personnes physiques qui ont créé l’Oeuvre.

5. “Offrant” : la ou les personne(s) physique(s) ou morale(s) qui proposent la mise à disposition de l’Oeuvre selon les termes du présent Contrat.

6. “Acceptant” : la personne physique ou morale qui accepte le présent contrat et exerce des droits sans en avoir violé les termes au préalable ou qui a reçu l’autorisation expresse de l’Offrant d’exercer des droits dans le cadre du présent contrat malgré une précédente violation de ce contrat.

Exceptions aux droits exclusifs

Aucune disposition de ce contrat n’a pour intention de réduire, limiter ou restreindre les prérogatives issues des exceptions aux droits, de l’épuisement des droits ou d’autres limitations aux droits exclusifs des ayants droit selon le droit de la propriété littéraire et artistique ou les autres lois applicables.

Autorisation

Soumis aux termes et conditions définis dans cette autorisation, et ceci pendant toute la durée de protection de l’Oeuvre par le droit de la propriété littéraire et artistique ou le droit applicable, l’Offrant accorde à l’Acceptant l’autorisation mondiale d’exercer à titre gratuit et non exclusif les droits suivants :

1. reproduire l'Oeuvre, incorporer l'Oeuvre dans une ou plusieurs Oeuvres dites Collectives et reproduire l'Oeuvre telle qu'incorporée dans lesdites Oeuvres dites Collectives;

2. distribuer des exemplaires ou enregistrements, présenter, représenter ou communiquer l'Oeuvre au public par tout procédé technique, y compris incorporée dans des Oeuvres Collectives;

3. lorsque l'Oeuvre est une base de données, extraire et réutiliser des parties substantielles de l'Oeuvre.

Les droits mentionnés ci-dessus peuvent être exercés sur tous les supports, médias, procédés techniques et formats. Les droits ci-dessus incluent le droit d'effectuer les modifications nécessaires techniquement à l'exercice des droits dans d'autres formats et procédés techniques. L'exercice de tous les droits qui ne sont pas expressément autorisés par l'Offrant ou dont il n'aurait pas la gestion demeure réservé, notamment les mécanismes de gestion collective obligatoire applicables décrits à l'article 4(d).

Restrictions

L'autorisation accordée par l'article 3 est expressément assujettie et limitée par le respect des restrictions suivantes :

1. L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public l'Oeuvre y compris par voie numérique uniquement selon les termes de ce Contrat. L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat à toute reproduction ou enregistrement de l'Oeuvre que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation de l'Oeuvre qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire. L'Acceptant ne peut pas céder de droits sur l'Oeuvre. L'Acceptant doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'exonération de responsabilité. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public l'Oeuvre, y compris par voie numérique, en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Oeuvre telle qu'incorporée dans une Oeuvre dite Collective, mais, en dehors de l'Oeuvre en elle-même, ne soumettent pas l'Oeuvre dite Collective,

aux termes du présent Contrat. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Auteur, comme demandé.

2. L'Acceptant ne peut exercer aucun des droits conférés par l'article 3 avec l'intention ou l'objectif d'obtenir un profit commercial ou une compensation financière personnelle. L'échange de l'Oeuvre avec d'autres Oeuvres protégées par le droit de la propriété littéraire et artistique par le partage électronique de fichiers, ou par tout autre moyen, n'est pas considéré comme un échange avec l'intention ou l'objectif d'un profit commercial ou d'une compensation financière personnelle, dans la mesure où aucun paiement ou compensation financière n'intervient en relation avec l'échange d'Oeuvres protégées.

3. Si l'Acceptant reproduit, distribue, représente ou communique l'Oeuvre au public, y compris par voie numérique, il doit conserver intactes toutes les informations sur le régime des droits et en attribuer la paternité à l'Auteur Original, de manière raisonnable au regard au médium ou au moyen utilisé. Il doit communiquer le nom de l'Auteur Original ou son éventuel pseudonyme s'il est indiqué ; le titre de l'Oeuvre Originale s'il est indiqué ; dans la mesure du possible, l'adresse Internet ou Identifiant Uniforme de Ressource (URI), s'il existe, spécifié par l'Offrant comme associé à l'Oeuvre, à moins que cette adresse ne renvoie pas aux informations légales (paternité et conditions d'utilisation de l'Oeuvre). Ces obligations d'attribution de paternité doivent être exécutées de manière raisonnable. Cependant, dans le cas d'une Oeuvre dite Collective, ces informations doivent, au minimum, apparaître à la place et de manière aussi visible que celles à laquelle apparaissent les informations de même nature.

4. Dans le cas où une utilisation de l'Oeuvre serait soumise à un régime légal de gestion collective obligatoire, l'Offrant se réserve le droit exclusif de collecter ces redevances par l'intermédiaire de la société de perception et de répartition des droits compétente. Sont notamment concernés la radiodiffusion et la communication dans un lieu public de phonogrammes publiés à des fins de commerce, certains cas de retransmission par câble et satellite, la copie privée d'Oeuvres fixées sur phonogrammes ou vidéogrammes, la reproduction par reprographie.

Garantie et exonération de responsabilité

1. En mettant l'Oeuvre à la disposition du public selon les termes de ce Contrat, l'Offrant déclare de bonne foi qu'à sa connaissance et dans les limites d'une enquête raisonnable :

   (a) L'Offrant a obtenu tous les droits sur l'Oeuvre nécessaires pour pouvoir autoriser l'exercice des droits accordés par le présent Contrat, et permettre la jouissance paisible et l'exercice licite de ces droits, ceci sans que l'Acceptant n'ait aucune obligation de verser de rémunération ou tout autre paiement ou droits, dans la limite des mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e);

   (b) L'Oeuvre n'est constitutive ni d'une violation des droits de tiers, notamment du droit de la propriété littéraire et artistique, du droit des marques, du droit de l'information, du droit civil ou de tout autre droit, ni de diffamation, de violation de la vie privée ou de tout autre préjudice délictuel à l'égard de toute tierce partie.

2. A l'exception des situations expressément mentionnées dans le présent Contrat ou dans un autre accord écrit, ou exigées par la loi applicable, l'Oeuvre est mise à disposition en l'état sans garantie d'aucune sorte, qu'elle soit expresse ou tacite, y compris à l'égard du contenu ou de l'exactitude de l'Oeuvre.

Limitation de responsabilité

A l'exception des garanties d'ordre public imposées par la loi applicable et des réparations imposées par le régime de la responsabilité vis-à-vis d'un tiers en raison de la violation des garanties prévues par l'article 5 du présent contrat, l'Offrant ne sera en aucun cas tenu responsable vis-à-vis de l'Acceptant, sur la base d'aucune théorie légale ni en raison d'aucun préjudice direct, indirect, matériel ou moral, résultant de l'exécution du présent Contrat ou de l'utilisation de l'Oeuvre, y compris dans l'hypothèse où l'Offrant avait connaissance de la possible existence d'un tel préjudice.

Résiliation

1. Tout manquement aux termes du contrat par l'Acceptant entraîne la résiliation automatique du Contrat et la fin des droits qui en découlent. Cependant, le contrat conserve ses effets envers les personnes physiques

ou morales qui ont reçu de la part de l'Acceptant, en exécution du présent contrat, la mise à disposition d'Oeuvres dites Dérivées, ou d'Oeuvres dites Collectives, ceci tant qu'elles respectent pleinement leurs obligations. Les sections 1, 2, 5, 6 et 7 du contrat continuent à s'appliquer après la résiliation de celui-ci.

2. Dans les limites indiquées ci-dessus, le présent Contrat s'applique pendant toute la durée de protection de l'Oeuvre selon le droit applicable. Néanmoins, l'Offrant se réserve à tout moment le droit d'exploiter l'Oeuvre sous des conditions contractuelles différentes, ou d'en cesser la diffusion; cependant, le recours à cette option ne doit pas conduire à retirer les effets du présent Contrat (ou de tout contrat qui a été ou doit être accordé selon les termes de ce Contrat), et ce Contrat continuera à s'appliquer dans tous ses effets jusqu'à ce que sa résiliation intervienne dans les conditions décrites ci-dessus.

Divers

1. A chaque reproduction ou communication au public par voie numérique de l'Oeuvre ou d'une Oeuvre dite Collective par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition de l'Oeuvre dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.

2. La nullité ou l'inapplicabilité d'une quelconque disposition de ce Contrat au regard de la loi applicable n'affecte pas celle des autres dispositions qui resteront pleinement valides et applicables. Sans action additionnelle par les parties à cet accord, lesdites dispositions devront être interprétées dans la mesure minimum nécessaire à leur validité et leur applicabilité.

3. Aucune limite, renonciation ou modification des termes ou dispositions du présent Contrat ne pourra être acceptée sans le consentement écrit et signé de la partie compétente.

4. Ce Contrat constitue le seul accord entre les parties à propos de l'Oeuvre mise ici à disposition. Il n'existe aucun élément annexe, accord supplémentaire ou mandat portant sur cette Oeuvre en dehors des éléments mentionnés ici. L'Offrant ne sera tenu par aucune disposition supplémentaire qui pourrait apparaître dans une quelconque communication en provenance de l'Acceptant. Ce Contrat ne peut être modifié sans l'accord mutuel écrit de l'Offrant et de l'Acceptant.

5. Le droit applicable est le droit français.

A.2 Creative Commons

Creative Commons n'est pas partie à ce Contrat et n'offre aucune forme de garantie relative à l'Oeuvre. Creative Commons décline toute responsabilité à l'égard de l'Acceptant ou de toute autre partie, quel que soit le fondement légal de cette responsabilité et quel que soit le préjudice subi, direct, indirect, matériel ou moral, qui surviendrait en rapport avec le présent Contrat. Cependant, si Creative Commons s'est expressément identifié comme Offrant pour mettre une Oeuvre à disposition selon les termes de ce Contrat, Creative Commons jouira de tous les droits et obligations d'un Offrant.

A l'exception des fins limitées à informer le public que l'Oeuvre est mise à disposition sous CPCC, aucune des parties n'utilisera la marque "Creative Commons" ou toute autre indication ou logo afférent sans le consentement préalable écrit de Creative Commons. Toute utilisation autorisée devra être effectuée en conformité avec les lignes directrices de Creative Commons à jour au moment de l'utilisation, telles qu'elles sont disponibles sur son site Internet ou sur simple demande.

Creative Commons peut être contacté à `http://creativecommons.org/`.