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

Benoît Groz

XML Security Views: Queries, Updates and Schemas

Thèse soutenue le 5 octobre 2012 devant le jury composé de :

Thomas Schwentick	Technische Universität Dortmund	Rapporteur
Michael Rusinowitch	INRIA Nancy	Rapporteur
Jean-François Raskin	Université Libre de Bruxelles	Examinateur
Hélène Touzet	CNRS	Présidente du Jury
Michael Benedikt	Oxford University	Invité
Sophie Tison	Université de Lille 1	Directrice
Sławomir Staworko	Université de Lille 3	Co-Encadrant

Summary: The evolution of web technologies and social trends fostered a shift from traditional enterprise databases to web services and online data. While making data more readily available to users, this evolution also raises additional security concerns regarding the privacy of users and more generally the disclosure of sensitive information. The implementation of appropriate access control models is one of the approaches to mitigate the threat. We investigate an access control model based on (non-materialized) XML views, as presented among others in [FCG04]. The simplicity of such views, and in particular the absence of arithmetic features and restructuring, facilitates their modelization with tree alignments. Our objective is therefore to investigate how to manipulate efficiently such views, using formal methods, and especially query rewriting and tree automata.

Our research follows essentially three directions: we first develop new algorithms to assess the expressivity of views, in terms of determinacy, query rewriting and certain answers. We show that those problems, although undecidable in our most general setting, can be decided under reasonable restrictions. Then we address the problem of handling updates in the security view framework. And last, we investigate the classical issues raised by schemata, focusing on the specific "determinism" requirements of DTDs and XML Schemata. In particular, we survey some techniques to approximate the set of all possible view documents with a DTD, and we provide new algorithms to check if the content models of a DTD are deterministic.

Résumé: Les évolutions technologiques ont consacré l'émergence des services web et du stockage des données en ligne, en complément des bases de données traditionnelles. Ces évolutions facilitent l'accès aux données, mais en contrepartie soulèvent de nouvelles problématiques de sécurité. La mise en œuvre de politiques de contrôle d'accès appropriées est une des approches permettant de réduire ces risques. Nous étudions ici les politiques de contrôle d'accès au niveau d'un document XML, politiques que nous modélisons par des vues de sécurité XML (non matérialisées) à l'instar de Fan et al. Ces vues peuvent être représentées facilement par des alignements d'arbres grâce à l'absence d'opérateurs arithmétiques ou de restructuration. Notre objectif est par conséquent d'examiner comment manipuler efficacement ce type de vues, à l'aide des méthodes formelles, et plus particulièrement des techniques de réécriture de requêtes et la théorie des automates d'arbres. Trois directions principales ont orienté nos recherches: nous avons tout d'abord élaboré des algorithmes pour évaluer l'expressivité d'une vue, en fonction des requêtes qui peuvent être exprimées à travers cette vue. Il s'avère que l'on ne peut décider en général si une vue permet d'exprimer une requête particulière, mais cela devient possible lorsque la vue satisfait des hypothèses générales. En second lieu, nous avons considéré les problèmes soulevés par la mise à jour du document à travers une vue. Enfin, nous proposons des solu-

tions pour construire automatiquement un schéma de la vue. En particulier, nous présentons différentes techniques pour représenter de façon approchée l'ensemble des documents au moyen d'une DTD.

Motivations

Contexte général Le projet résumé dans ce manuscript a pour but de développer des techniques inspirées par les méthodes formelles pour manipuler des vues de sécurité XML. XML s'est établi depuis une dizaine d'années comme le format par excellence pour l'échange de données, et, dans une moindre mesure, pour la publication de données sur le Web. Les évolutions technologiques de ces dernières années ont consacré l'émergence des services web et la mise en ligne des données en complément des bases de données internes plus "traditionnelles". Ces nouvelles modalités de stockage et d'accès aux données soulèvent la question de la sécurité des données. Les techniques de contrôle d'accès couvrent un aspect essentiel de la sécurité informatique, en veillant à préserver la confidentialité et l'intégrité des informations. Dans le cadre de cette thèse nous considérons des problématiques de contrôle d'accès au niveau du document. La politique de contrôle d'accès est modélisée par une vue de sécurité non-matérialisée. Pour ces vues, inspirées du modèle de Fan et al. [FCG04, FGJK07], la partie du document accessible à l'utilisateur – que nous désignerons par *document de vue*– n'est pas matérialisée, et l'on se contente de calculer la réponse aux requêtes de l'utilisateur. Nous proposons des solutions pour comparer deux vues, mettre à jour le document à travers une vue, ou encore construire un schéma qui représente l'ensemble des documents de vues possibles.

Modélisation par des langages d'arbres Un document XML est formé d'une suite de balises ouvrantes et fermantes, qui doivent être bien imbriquées. Ceci permet de modéliser chaque document XML par un arbre, comme illustré en Figure 1 (le document XML a été tronqué par souci de lisibilité). De nombreux langages ont été définis et standardisés pour faciliter la manipulation de documents au format XML, qu'il s'agisse de langages de requêtes comme XPath et XQuery, ou bien de langages de schémas comme les DTDs et XML Schema. Nos travaux s'appuient sur ces langages de la constellation XML et sur la théorie des automates pour modéliser les vues, requêtes et mises à jour.

En particulier, une vue de sécurité est définie essentiellement comme une paire formée d'une DTD et d'une requête XPath. La DTD représente l'ensemble des formes que peut prendre le document, et la requête associe à un document l'ensemble de ses éléments accessibles à l'utilisateur, comme illustré dans l'exemple 0.1.

Nous considérons principalement deux formalismes pour définir des requêtes:

```
<projets>
  <projet>
    <nom>
    <stable>
      <src></src>
      <bin></bin>
      <doc></doc>
    </stable>
    <licence>
         <libre>
         </libre>
    </licence>
  </projet>
  ...
</projets>
```

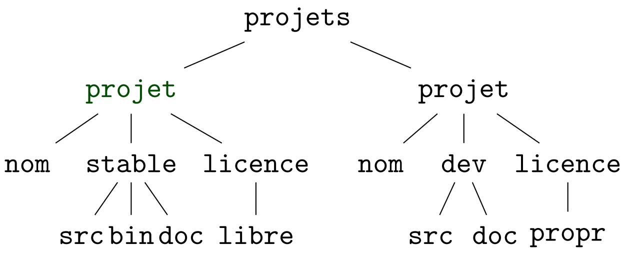


Figure 1.: Représentation arborescente d'un document XML t_0.

les expressions Regular XPath, qui étendent le fragment navigationnel de XPath 1.0 avec un opérateur de clôture transitive, et les automates d'arbres. Plus précisément nous employons des automates visibly pushdown (VPA), définis par Alur et Madhusudan [AM04b] et particulièrement bien adaptés à la représentation de langages d'arbres d'arité non bornée. Ces automates définissent par défaut des requêtes booléennes (caractérisées par le langage accepté par l'automate). Pour définir des requêtes plus générales, ainsi que des transformations de documents comme les vues et les mises à jour, nous étudions des langages réguliers d'arbres d'alignements, c'est-à-dire des arbres sur des alphabets de la forme $\Sigma \times \Sigma \cup \{\varepsilon\} \times \Sigma \cup \Sigma \times \{\varepsilon\}$, où Σ est l'alphabet du document et ε un symbole spécial. Un élément étiqueté par (a, ε), par exemple, correspond à l'effacement d'un noeud étiqueté a dans l'arbre d'entrée, alors qu'un noeud étiqueté par (a, a) (resp. (a, b)) est préservé tel quel (resp. renommé en b).

Example 0.1. *La DTD D_0 ci-dessous décrit un ensemble de projets informatiques. Chaque projet a un nom, une licence, et peut être soit stable soit en cours développement (*`dev`*). Un projet en cours de développement contient des sources (*`src`*) et de la documentation (*`doc`*). Un projet stable contient en outre des fichiers binaires (regroupés sous* `bin`*). La licence d'un projet peut-être soit libre (*`libre`*) soit propriétaire (*`propr`*). Le document de la figure 1 satisfait cette DTD D_0.*

L'annotation $\mathtt{ann}_0$ donne accès à tous les projets, mais cache le statut (développement ou stable) des projets, et en particulier cache les fichiers binaires. En outre, les sources sont cachées pour les projets sous licence propriétaire. Lorsque la visibilité d'un élément n'est pas spécifiée, elle est héritée du plus proche ancêtre pour lequel elle est spécifiée, et la racine (`projects`*) est toujours visible par défaut.*

$\texttt{projects} \to \texttt{project*}$
$\texttt{project} \to \texttt{name}, (\texttt{stable} \mid \texttt{dev}), \texttt{license}$
$\quad \mathsf{ann}_0(\texttt{project}, \texttt{stable}) = \mathsf{false}$
$\quad \mathsf{ann}_0(\texttt{project}, \texttt{dev}) = \mathsf{false}$
$\texttt{license} \to \texttt{libre} \mid \texttt{propr}$

$\texttt{stable} \to \texttt{src}, \texttt{bin}, \texttt{doc}$
$\quad \mathsf{ann}_0(\texttt{stable}, \texttt{src}) = [\Uparrow^*\text{::}\texttt{project}/\Downarrow^*\text{::}\texttt{libre}]$
$\quad \mathsf{ann}_0(\texttt{stable}, \texttt{doc}) = \mathsf{true}$
$\texttt{dev} \to \texttt{src}, \texttt{doc}$
$\quad \mathsf{ann}_0(\texttt{dev}, \texttt{src}) = [\Uparrow^*\text{::}\texttt{project}/\Downarrow^*\text{::}\texttt{libre}]$
$\quad \mathsf{ann}_0(\texttt{dev}, \texttt{doc}) = \mathsf{true}$

La figure 2 représente l'arbre d'alignement entre le document t_0 et sa vue, tandis que la figure 3 présente l'arbre de vue résultant.

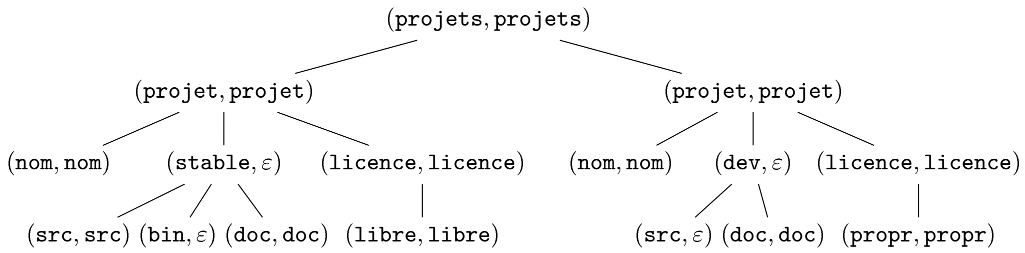


Figure 2.: Arbre d'alignement entre t_0 et sa vue pour (D_0, ann_0)

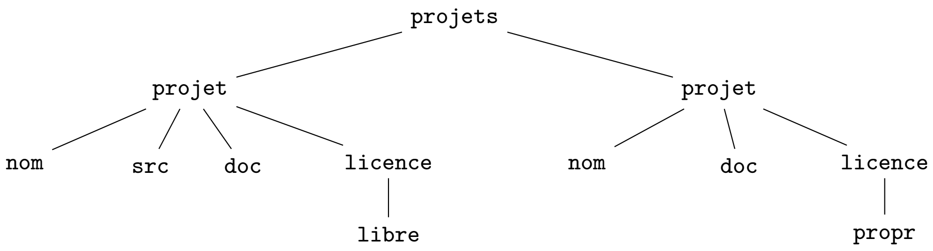


Figure 3.: Vue de t_0 pour (D_0, ann_0)

Contributions

Les problèmes que nous étudions et les réponses que nous apportons peuvent être résumés comme suit.

Évaluation des requêtes sur des vues non-matérialisées Tout d'abord le modèle de vue non-matérialisée impose de calculer la réponse aux requêtes de l'utilisateur directement à partir du document original (qui contient les parties inaccessibles à l'utilisateur), alors même que la requête de l'utilisateur est formulée sur le schéma de vue. Ceci impose de reformuler la requête de l'utilisateur en fonction du document original (voir figure 4, point (3)). Nous montrons que cette composition de la requête de vue avec la requête de l'utilisateur peut être obtenue efficacement pour des requêtes Regular XPath, ce qui n'est pas le cas pour de nombreux fragments XPath [FCG04, FGJK07, VHP06]. Cette approche se distingue du modèle de Fan et al. sous deux aspects: d'une part il n'est pas nécessaire d'exploiter les schémas du document original et de la vue pour calculer la requête à appliquer: celle-ci peut-être définie directement à partir de la requête de l'utilisateur et d'une formule XPath décrivant la vue, et d'autre part la requête obtenue par l'algorithme de composition est directement une requête XPath au lieu d'un modèle d'automate ad-hoc dans [FGJK07]. De ce point de vue, l'utilisation de Regular XPath en incluant tous les axes (axes descendants et ascendants, ainsi que les axes horizontaux) simplifie le processus de réécriture, ainsi que la définition des vues, en permettant de représenter la politique par une requête unique. Par contre l'expressivité des vues ainsi obtenues soulève de nouveaux problèmes pour la construction du schéma de vue, et rend potentiellement plus difficile l'optimisation des requêtes. Pour des requêtes et vues exprimées par VPAs, la composition peut aussi être calculée en temps polynomial, par une construction standard d'automate produit.

Mises à jour sur des vues non-matérialisées La même question se pose pour les mises à jour: comme l'utilisateur formule ses mises à jour sur le schéma de vue, celles-ci ne peuvent pas en général être appliquées directement au document original. La littérature désigne par *view update problem* (ou mises à jour à travers des vues) le problème consistant à calculer la (une) mise à jour qui doit être appliquée au document source pour obtenir l'effet souhaité par l'utilisateur sur la vue: plus formellement, il s'agit d'obtenir la mise à jour pour que le diagramme en figure 5 soit commutatif.

Nous étudions en particulier le problème de mise à jour à travers une vue lorsque la mise à jour est spécifiée par une fonction associant à chaque document de vue possible le document attendu après la mise à jour. Nous représentons de telles fonctions par un ensemble régulier d'arbres d'alignement, en imposant la contrainte supplémentaire que les insertions et suppressions doivent concerner des sous-arbres complets (i.e., il n'est pas possible de supprimer (resp. d'insérer) un noeud sans supprimer (resp. insérer) aussi tous ses descendants. Notons toutefois que les opérations atomiques du langage de mise à jour XQUF (XQuery Update Facility) du W3C ne permettent elles aussi les insertions et suppressions qu'au niveau des feuilles d'un document. En l'absence de contraintes particulières, cette formulation du problème de mise à jour à travers une vue admet une solution polynomiale. En revanche, lorsque l'ensemble des mises à jour autorisées sur le document est restreint à un ensemble régulier d'alignements, le problème soulève de nouvelles difficultés que nous étudions en détail.

Approximation du schéma de vue Dans le modèle de Fan, il est possible d'interdire à l'utilisateur l'accès à des noeuds internes tout en conservant leurs descendants dans la vue. En général, ces noeuds internes restent alors présents dans la vue; seule leur étiquette est anonymisée. À l'instar de Kuper et al., nous adoptons au contraire la sémantique qui consiste à supprimer complètement ces noeuds, en faisant adopter un noeud par son plus proche ancêtre accessible. Ce choix complique la construction d'un schéma de vue. Il est bien entendu aussi possible d'anonymiser une étiquette, mais ceci se fait seulement par l'opération de renommage, et non par des suppressions. Lorsque le schéma décrivant l'ensemble des documents originaux possibles est une DTD non-récursive, et même pour des vues très simples, l'ensemble des documents de vue possibles peut définir des langages d'arbres non-réguliers: il devient essentiellement nécessaire d'utiliser des grammaires algébriques pour décrire le schéma de vue.

La première approche que nous suggérons pour limiter l'expressivité du schéma de vue est d'imposer des restrictions sur la vue. Ces restrictions permettent aussi de faciliter les autres problèmes comme la comparaison de politiques ou la vérification de propriétés sur les mises à jour de documents. La première restriction est d'imposer que les vues soient closes vers le haut: un noeud ne peut alors être visible que si tout ses ancêtres aussi le sont. Cette restriction apparait fréquemment dans la littérature sur les vues XML. Une autre restriction courante consiste à borner la profondeur du document original par une constante. Cette hypothèse peut aussi sembler raisonnable du fait que la plupart des documents XML observés sur le web ont une faible profondeur [BMV06] mais cette restriction exclut les DTDs récursives. Nous proposons une troisième approche, moins restrictive que les deux précédentes. Cette contrainte que nous appelons k-interval-boundedness impose une constante k bornant pour tout chemin d'une feuille jusqu'à la racine, le nombre de noeuds internes consécutifs que l'on efface sur ce chemin. Pour toute vue V spécifiée par un VPA ou une formule de Regular XPath, s'il existe une constante k telle que V est k-interval-bounded, alors le schéma de vue pour V est un ensemble régulier d'arbres.

De notre point de vue, le schéma de vue sert essentiellement pour permettre à l'utilisateur de formuler ses requêtes, et c'est pourquoi nous nous intéressons au problème d'approximer le schéma de vue dans cette perspective. L'approximation du schéma de vue est compliquée par une contrainte spécifique aux schémas XML; à savoir que les expressions régulières qui apparaissent dans ces schémas doivent être des expressions régulières déterministes. Nous montrons en particulier que l'on peut tester en temps linéaire si une expression régulière est déterministe ou non, même en présence d'indicateurs numériques, un problème ouvert formulé Kilpelaïnen and Tuhkanen [KT07, Kil11]. Pour les cas où l'on doit recourir à une approximation, nous étudions trois techniques permettant d'approximer une grammaire algébrique par un langage régulier.

Comparaison de politiques Un problème classique lorsque l'on utilise des vues est d'étudier leur expressivité: quelle information peut-être extraite à partir d'une vue donnée? Dans un contexte de sécurité il importe de vérifier qu'un adversaire ayant accès à la vue ne peut pas accéder à une information jugée confidentielle. Nous proposons ainsi des méthodes pour comparer des politiques de sécurité. De telles méthodes pourraient typiquement servir dans un scénario où l'administrateur choisirait de modifier la politique, et souhaiterait vérifier que la nouvelle vue ne permet pas d'inférer des informations qui étaient confidentielles avant la modification. Nous observons que ce problème de comparaison de politiques peut être relié au problème de décider si une requête Q est déterminée par une vue V; est-ce que la connaissance de $\mathcal{V}\mathit{iew}\,(V, t)$ (la vue de t pour V) suffit à calculer $Q(t)$ pour tout document t? Cette question est indécidable pour des requêtes et vues définies par des VPAs ou formules XPath arbitraires, mais nous proposons des algorithmes répondant à cette question en temps au plus exponentiel pour des vues k-interval bounded.

Perspectives

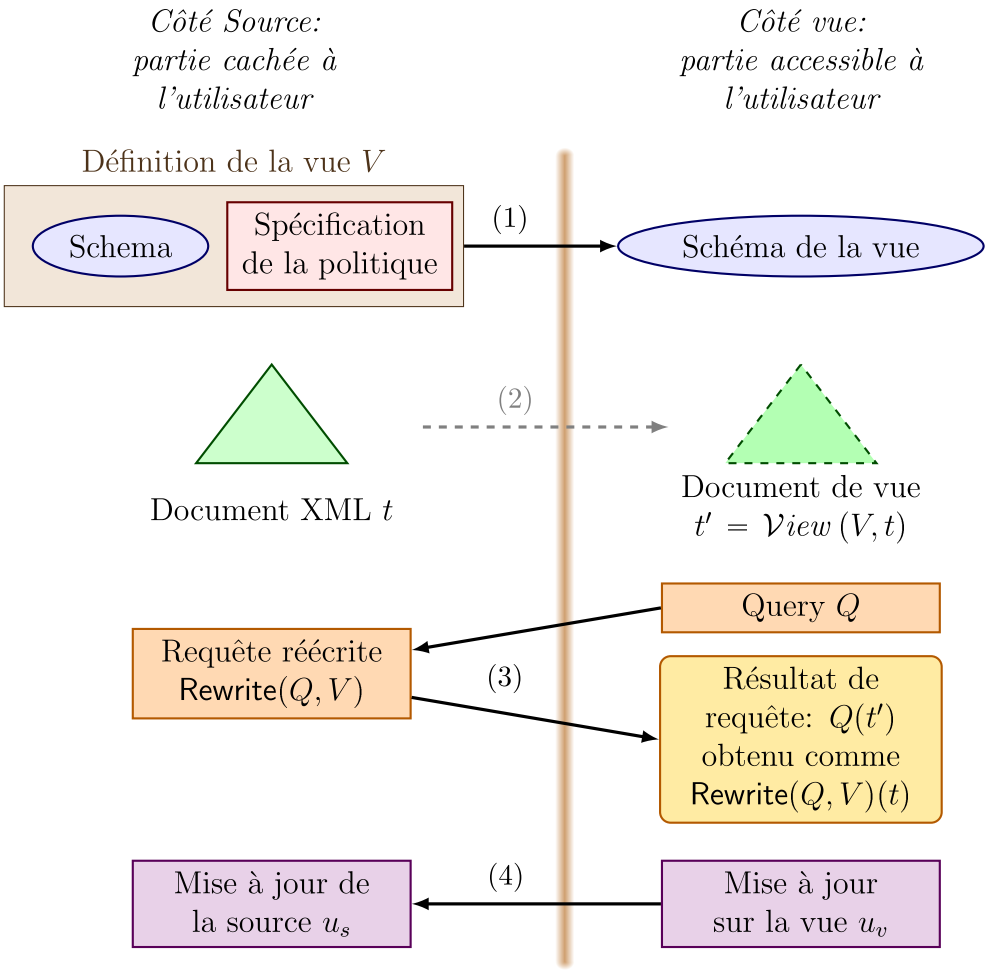
Les travaux résumés dans cette thèse peuvent être étendus dans plusieurs directions.

Des modèles plus expressifs Les vues, requêtes et mises à jour étudiées dans ce document sont limitées à des langages réguliers d'arbres ou au fragment navigationnel du langage XPath. Ceci peut sembler trop restrictif pour de nombreuses applications, en particulier pour prendre en compte les méchanismes de clefs omniprésents dans les bases de données, ou les opérations d'aggrégation. On pourrait envisager d'utiliser d'autres modèles, peut-être basés sur les logiques prenant en compte les données. On pourrait aussi envisager d'utiliser pour les vues d'autres modèles de transducteurs permettant de copier des parties du document, et, plus généralement, de réorganiser le document.

Par ailleurs, et dans la mesure ou les bases de données XML natives n'ont pas connu un grand succès, il serait intéressant d'appliquer les techniques de query rewriting sur des modèles de graphes au lieu d'arbres. Bien sûr, de tels travaux ont déjà été entrepris dans cette direction, par exemple sur la réécriture de requêtes conjonctives [NSV10, Pas11] ou de regular path queries [CGLV02, CGLV07], mais de nombreux problèmes restent ouvert.

Optimisations pour les VPAs La figure 6.7 montre que les automates visibly pushdowns ont suscité un intérêt croissant depuis leur introduction par Alur et Madhusudan. Nous pensons qu'il reste plusieurs sujets dignes d'intérêt pour mieux comprendre ce modèle d'automate. À commencer par l'optimisation des opérations fondamentales telle l'évaluation d'un VPA sur un document (tout particulièrement pour les modèles de VPAs définissant des requêtes au lieu de langages booléens). Un des problèmes sous-jacents est la question de traiter efficacement le non-déterminisme dans les VPAs, le non-déterminisme soulevant des défis pour les VPAs comme pour la plupart des modèles d'automates d'arbres, et a fortiori de transducteurs.

Optimisations pour les langages de schémas XML La restriction à des expressions déterministes dans les DTD et Schémas XML est sujette à discussions [Man01, W3C]. Néanmoins les algorithmes actuels de validation de schéma n'exploitent pas complètement le déterminisme des expressions pour optimiser les performances. Nous avons définis des algorithmes radicalement nouveaux pour manipuler des expressions déterministes, mais il n'est pas clair si ces algorithmes permettraient des gains significatifs sur des schémas réels. Nous avons l'intention d'évaluer expérimentalement les performances de ces algorithmes. Par ailleurs, ces algorithmes soulèvent la question de la complexité exacte des problèmes de décision pour les expressions déterministes. Qui plus est, il serait intéressant d'étudier si les techniques développées dans le cadre des schémas XML peuvent trouver une application dans d'autres domaines.


Côté Source: partie cachée à l'utilisateur
Côté vue: partie accessible à l'utilisateur
Définition de la vue V
Schema
Spécification de la politique
(1)
Schéma de la vue
(2)
Document XML t
Document de vue $t' = \mathcal{V}iew\,(V, t)$
Query Q
Requête réécrite $\mathsf{Rewrite}(Q, V)$
(3)
Résultat de requête: $Q(t')$ obtenu comme $\mathsf{Rewrite}(Q, V)(t)$
Mise à jour de la source u_s
(4)
Mise à jour sur la vue u_v

(1): Construction du Schéma de vue (et approximation)

(2): Matérialization (virtuelle)

(3): Évaluation des requêtes (par composition avec la vue)

(4): Traduction des mises à jour

Figure 4.: Vues de sécurité non-matérialisées.

$$\begin{tikzcd} {} \arrow[r, dotted, "\text{traduction } u_s \text{ de la mise à jour}"] \arrow[d, "V"'] & {} \arrow[d, "V"] \\ {} \arrow[r, "\text{mise à jour } u_v"'] & {} \end{tikzcd}$$

$$u_v = V^{-1} \circ u_s \circ V$$

Figure 5.: View update problem.

Thanks

I will never be too grateful to all who offered me help, advice and friendship during this PhD. My thanks go first to my loving family, to whom I owe so much that I shan't write it down (this dissertation is already long enough).

Michaël Rusinowitch and Thomas Schwentick did an impressive job to review this dissertation. I am very obliged to them for their very thorough examination of this work, appreciations and suggestions. Michael Benedikt also agreed to review the dissertation and contributed many references and other advice. In addition to their tremendous work at reviewing those 300 pages, I appreciate their patience waiting for the long-overdue dissertation. I would also like to thank Jean-François Raskin and Hélène Touzet for accepting to take part in the jury of my defense.

It was a pleasure to work under the supervision of Sophie and Slawek, devoted teachers and researchers always eager to contribute new questions or techniques, proof ideas, or to check my own proofs. May I some day match their leadership and scholarship. I am especially thankful that these supervisors shared so generously their time with me, always willing to discuss any topic I felt interested in. I would in particular like to express my gratitude to Slawek for taking the thankless charge of improving my writing skills.

I have also been very fortunate to work with Yves, who helped shape up this PhD, contributed several of the results, and shared his great knowledge of language theory. Many thanks to Anne-Cécile, Iovka, and Yves for their contributions in our project: they were delightful colleagues to work with.

Working within the Mostrare team was a very gratifying experience. Its talented members are engaged in very diverse research topics and yet have kept it a merry, united, and very lively team.

Joachim in particular communicated his great enthusiasm for research along many discussions. Among others, I owe him numerous insights on VPAs and tree automata, and he offered me a very nice opportunity to collaborate with Sebastian. Sebastian also deserves my gratitude for his knack of finding interesting research topics, his patience when confronted with my sketchy proofs, and for his very warm welcome in Sydney.

This dissertation and project owe a lot to Olivier, through his support, his introducing me to VPAs, and his checking a part of this dissertation. Sharing office with Guillaume and Antoine is the guarantee of having a great laugh at least once a day. Antoine also introduced us to very interesting questions about property testing, whereas I benefited from Guillaume's tremendous knowledge of computer theory and graphs. I have also enjoyed chatting with Tom and Denis about their thrilling project on streaming evaluation, and about a miscellany of other topics.

The very good humour of Jean-Baptiste, Adrien, and Fabien were further incentives to work harder toward the completion of my PhD. Without Marc I couldn't have overcome the crashing of my debian at the peak of the dissertation writing. May Antonino find a just retribution for trying to choke me with italian biscuits. And so may Grégoire and Jean for hopelessly trying to fatten me with sweets, Jérôme for numerous discussions and afternoons when I could appreciate his organizing skills, or Rémi, Angela, Aurélien and tutti quanti for pestering me about my dissertation in order to make sure I would someday complete the redaction.

Many thanks finally to the teachers and to the supervisors of my previous internships; Wolfgang, Jean-François and Luc, who communicated to me their interest for science and research. I also wish to thank the administrative staff, especially the INRIA team assistants for their invaluable help. I definitely admired the effort of many reviewers, especially at PODS, who helped correct, reformulate and simplify several of our results.

My last thanks go to all the friends that enlightened my student years: schoolmates, board game players, members of the C4, o.p., musicians, flatmates, and all those with whom I shared a few steps along the path of life...

Contents

1. Introduction

The whole difference between a construction and a creation is exactly this: that a thing constructed can be loved after it is constructed; but a thing created is loved before it exists.

(G. K. Chesterton)

Contents

1.1. Motivation

The project summarized in this dissertation aims at developing techniques to support access control over XML documents, a topic that raised considerable interest over the last few years. The ever-increasing role of the web in society both comforts the expansion of XML technologies and raises growing concerns about the security of data. The evolution of web technologies and social trends fostered a shift from traditional database management systems toward distributed storage of data and online services. With more and more data accessible from the web, preserving the confidentiality of sensitive information such as customer data has emerged as one of the main challenges for computer security. Applications raising security concerns span domains as diverse as media sharing, social networks, biological databases, healthcare systems and financial data. Independently of security considerations, views can also be used to extract and organize information. One major issue when considering views is the management of a dynamic environment: the policy security (the view specification) may evolve over time, and, of course, the document may be frequently updated. The support of update operations appears in two of the twelve (thirteen actually) rules of Codd specifying the requirements for a relational database management system: rule 6 requires the support of view update mechanisms (for updatable views), whereas rule 7 requires the support of update operations that manipulate sets instead of a single tuple. The security view model that inspired our work faces those challenges for XML databases by keeping the view virtual. Our work extends this model in terms of expressivity and support for view update operations. We also consider techniques to reason about views and updates.

XML, the lingua franca on the web(?) Over the last two decades, the Extensible Markup Language (XML) has evolved into a gold standard for representing and exchanging data. The W3C and some other organizations developed several specific schema and query languages to process XML documents, such as XML Schema, XPath, and XQuery. The XML Path Language (XPath) is the core of all these query languages to address the elements of the XML document. Three versions of XPath have been proposed successively by the W3C, but this dissertation only exploits features from XPath 1.0, and more accurately the navigational core of this language. So, whenever we mention XPath in this dissertation we refer to a subset of XPath 1.0. (generally extended with a transitive closure operator to form Regular XPath). While some databases store data into traditional database management systems (DBMS) and use XML only for exporting information, more and more DBMS provide an XQuery engine. Storing data in XML format avoids the conversion cost.

Securing the data: privacy and access control Access control encompasses mechanisms to specify and enforce a security policy that limits the actions a user can perform. Access control mechanisms permit individuals and organizations to share information while preserving the confidentiality and integrity of data according to the user's wishes. The *read, write* and *execute* permissions attached to the files in the Unix systems are a typical example of access control implementation that allows multiple users and programs executed on behalf of the user(s) to share resources on a computer. It is also worth observing that the big effort toward formalizing access control [Lam71, GD72, BL73] follows shortly after the commercialization of time-sharing systems, that flourished in the 1970's when multiple organizations shared the cost of leasing a computer [Bel05]. This also coincides with the development of computer networks.

Why should the access to data be controlled? A first reason to control the access to data could be to filter out irrelevant information. This argument is especially relevant for view-based access control models. Yet the foremost arguments for access control are privacy and security issues; to preserve the privacy of individuals and prevent the dissemination of sensitive information that could harm individuals or companies.

Access control mechanisms at the database level represent but a fraction of much broader security and privacy perspectives. Although our thesis remains focused on the specific view mechanism for access control at the database level, we briefly survey the impact of security and privacy on economy and politics. Privacy concerns triggered some legislative actions compelling institutions to implement policies preventing the disclosure of personal data. Prominent among privacy-aware legislation are the *Health Insurance Portability and Accountability Act* (HIPAA) and the *Gramm-Leach-Bliley Act* (GLB), enacted by the U.S. Congress in 1996 and 1999. The administrative simplification provisions of HIPAA address the privacy of health data and include substantial penalties for failures to comply with national standards and operating rules. The privacy provisions from the GLB act require financial institutions to provide their customers some notice before they disclose information to non-affiliated third parties. More generally, the U.S. Federal Trade Commission's Fair Information Practice Principles gives recommendations concerning data collection practices. The European Union's *Data Protection Directive* harmonizes the processing of personal data between the member states. A reform is under way in order to put an end to divergences in the enforcement of the previous directive between member states, strengthen privacy, simplify administrative requirements and take into account the evolution of digital economy since 1995 [EUd12]. These regulations clearly demonstrate the concern of governments and individuals for privacy protection. Let us however observe that while privacy and access control overlap, the preservation of privacy raises many questions that we do not consider in our access control model: our model provides no clue on how general statistics about medical records (averages,. . .) could be made available to scientists while preventing statistical inference of individual information about the patients in presence of an adversary armed with a priori information. Techniques to handle that setting often rely on differential privacy, a notion that lies outside the scope of this dissertation.

Improper access control implementations in a broad sense often hit the headlines. To mention but a few: an attack on Sony's PlayStation network in April 2011 compromised over 100 million customer accounts, including street numbers, email, and passwords [Son11]. On March 30th, an attack retrieved huge mailing lists from Epsilon, a leading online marketing company. On March 24th, tripadvisor informed its customers that part of the (reportedly) 20 million addresses it collects had been leaked in a database breach. On June 9th, CitiBank communicated a breach into 1% of its credit card accounts, caused by the possibility to access user accounts without authorization checks by modifying URLs. affecting some two hundred thousand customers. The latest large-scale security breach to date hit GlobalPayment, which reported on March 30th, 2012 that an estimated 1.500.000 card numbers may have been compromised as a result of unauthorized access into its processing system.

What kind of information is generally stolen, and which vulnerabilities are exploited? When trying to assess the extent of the threat, it seems that, beyond a miscellany of blogs from security experts discussing vulnerabilities, patches and data leaks, reports from private IT-security companies provide an interesting overview. The *2011 Data Breach Investigation Report* [DBI] by Verizon with the U.S. Secret Service, and Dutch National High Tech Crime Unit observes a steady drop in the annual number of compromised records since 2008: 361 millions in 2008, 144 millions in 2009, and 4 millions in 2010. The authors suggest the successful identification, prosecution and incarceration of the wrongdoers is the main explanation for this trend. A huge majority of those attacks stems from external agents and does not implicate insiders. The report also notes that criminals tend to turn away from big institutions, targeting most of the attacks at smaller target such as hospitality sectors and retail industries, the main victims of opportunistic attacks. The assets compromised are in most cases points of sale, database servers and web servers. Payment card numbers still account for most of the compromised records investigated in this Verizon record, followed by authentication credentials. But the authors think the focus may still continue to shift from payment card to other kinds of data such as personal information, although the loss of information appears in less than 15% of the incidents investigated, and in less than 1% of the compromised records. Those figures may seem impressive, yet they do not take into account those of the access control failures that do not register as criminal offence but still affect individuals. On the whole, the figures stress the need for better controlling the access to sensitive information, though it is hard to single out the impact of the document level mechanisms in this broad picture of access control. Regarding specific XML technology, we observe a few examples of breaches specific to xml processing. A major vulnerability was discovered in Microsoft's IE7 in 2008 [vul08], and vulnerabilities were discovered in XML libraries, prompting some experts to expect that XML-based attack would flourish soon [xml09]. There is no doubt, however, about access control being considered a crucial feature in database systems. According to the SANS report [SAN10], improper access control belongs to the 25 most dangerous software errors. More specifically, the 2010 report the inconsistency or absence of authorization as a highly prevalent weakness with high likelihood of being exploited. The 2011 report further distinguishes the absence of access control checks and their incorrect implementation.

Many general models for access control have been developed to formalize access control, and these models have been implemented in relational databases: the major database management systems such as Oracle 11g and IBM DB2 support fine-grained access control mechanisms. More recently, several access control mechanisms have been proposed to take into account the specifics of XML, namely the tree structure of the document, the specific query languages (XPath,XQuery) and the possibility to define a schema with a DTD (see, e.g., [FM04]). This thesis is thus part of a larger effort from the community to develop models for XML access control.

1.2. Security Views and Query rewriting as a model for Access Control

An access control policy defines which data should be accessible to the user and which should be kept hidden. A popular model for access control is *Role Based Access Control*, in which each user is assigned or may choose a role and the policy defines which actions are allowed for each role. Role based access control emphasizes the difference between the user's identity and the role. In general, the decision to grant or deny the execution of a query should not depend only on the user's identity, but also on the context of the query: the user may run some third-party software which he does not trust, etc. Furthermore, the policy may also take into account numerous elements such as time and physical location. In our framework we assume the role has been defined, and the data accessible to the user is represented as a (single) view. Thus, the information the user can obtain from the view is exactly that which he can obtain from the document using this role, according to the policy.

In the relational model, a view is generally a virtual relation, stored as a query to the original database. The user can then use the view relation to formulate queries, but the view relation needs not be materialized as a table as it can be computed on-the-fly. Of course commercial databases also allow to materialize the view, in which case the table for the view relation must be updated when the original database is modified. In NoSQL databases, and especially document-oriented databases, queries need not be formulated in terms of algebraic operations on relations stored as tables, but use specific query languages to extract information from collections of documents. In the XML framework, views (i.e., queries) are commonly expressed via XQuery or directly with XPath expressions, which define paths to access resources within a document. The evaluation of the view query returns a set of documents. We consider the case of a single XPath view on a single XML document, returning a single XML (tree-structured) document. In that case we call *view document* the document resulting from the view query evaluation.

Following the security view framework of Fan et al. [FCG04, FGJK07], we consider non-materialized views, meaning that the view is stored as a query, and the view document is not materialized. In this security view framework, the user has no direct access to the real document. Instead, each user (or role) is assigned a view, and the user has only access to the view document to query the database. For better efficiency, the view document is not materialized: the user is not provided the current view document $V(t)$, which is kept implicit, but instead she is only provided a schema of the view. Then, the user queries must be rewritten before they can be executed on the real document.

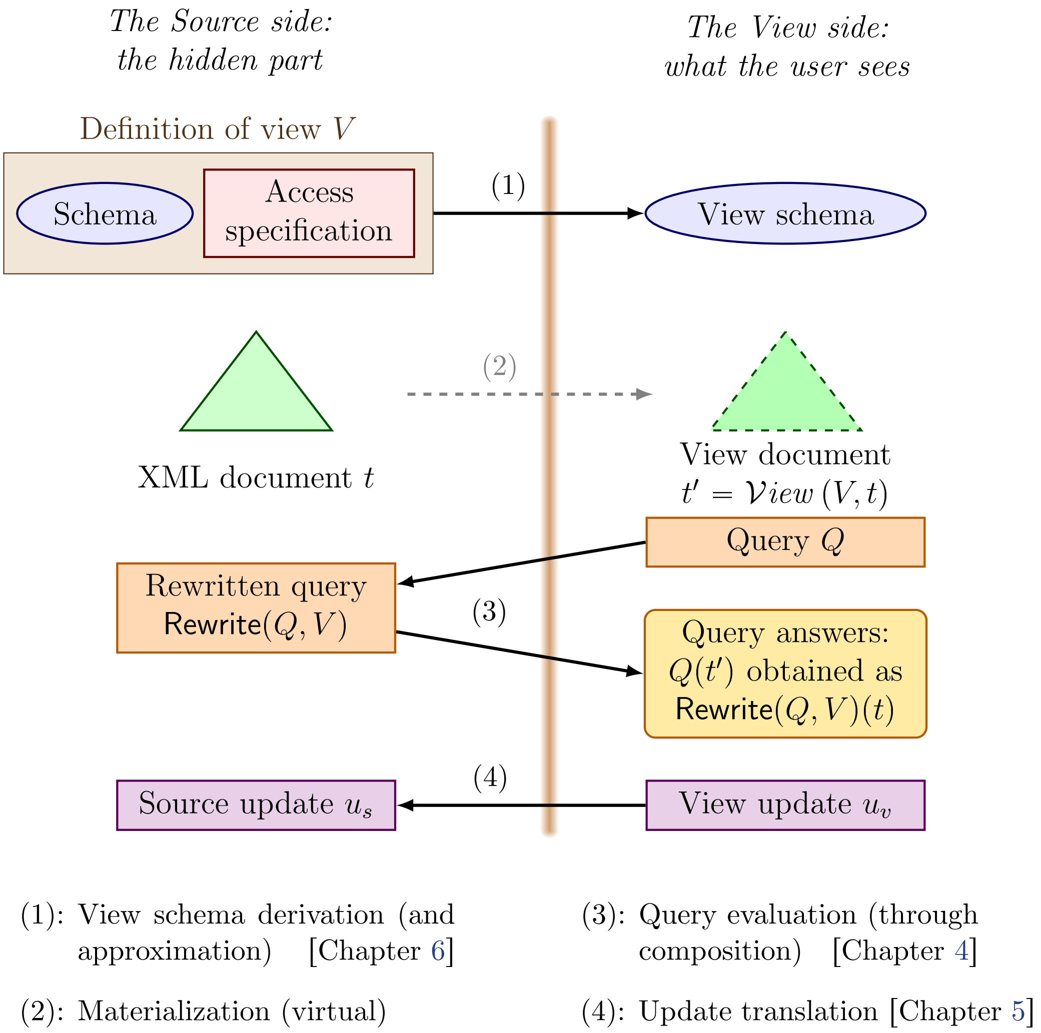

The Source side: the hidden part
The View side: what the user sees
Definition of view V
Schema
Access specification
(1)
View schema
(2)
XML document t
View document
$t' = \mathcal{V}iew\,(V, t)$
Query Q
Rewritten query
$\mathsf{Rewrite}(Q, V)$
(3)
Query answers:
$Q(t')$ obtained as
$\mathsf{Rewrite}(Q, V)(t)$
(4)
Source update u_s
View update u_v
(1): View schema derivation (and approximation) [Chapter 6]
(3): Query evaluation (through composition) [Chapter 4]
(2): Materialization (virtual)
(4): Update translation [Chapter 5]

Figure 1.1.: Non-materialized security views.

Specific questions addressed in the project The specific problems that we investigate can be summarized as follows. When the user formulates her query on the view, how can we reformulate the query to obtain the answers from the document (without first materializing the view) (cf. (3) in Figure 1.1)? The same question arises also for write queries, a.k.a. updates, and this problem of translating updates from the view into modifications of the real document is known as the *view update problem* (4). As the user needs some information on the view to formulate meaningful queries, one must provide algorithms to compute a schema for the set of all possible view documents. How can we represent the set of all possible view documents, in particular when this set must be approximated (1) ? When the access control policy is modified, how can we check if the new view discloses strictly less information than the original one? More generally, what information can be extracted from a view? And finally, since we assume that our views come with a schema, how can we efficiently validate a document against a schema? We outline in the following our approach to treat those questions, and the results obtained.

1.3. Our Contributions

We believe that formal methods provide powerful tools for reasoning about access control, and this dissertation presents our contributions in this field. We mostly interested ourselves in the evolution of database system, and propose methods to verify which properties are maintained when the document or the security policy are modified. Our study focuses on the query rewriting approach over non-materialized XML security views, but we actually address general issues, whose applications span far beyond this security view framework. Throughout this research journey, we have extended the security view framework of Fan et al., and contributed some answers to the problems of schema validation, policy comparison, and to the view update problem for XML.

A general model for security views Our first contribution is an extension to all axes from XPath of the query rewriting framework for security views as presented, among others, by Fan et al. [FGJK07]. Thus, our model can define more expressive access control policies while maintaining a quadratic complexity for querying the database. This increase in expressiveness raises a few issues at the database level. The query rewriting algorithm of Fan et al. relies on the DTD schema to rewrite the user's queries, but with our more expressive fragment the rewriting process does not need to rely on the schema anymore. Actually, the rewriting can be seen as the *composition* of two queries, a relatively simple task.

Schema approximation As in the model of Kuper et al. [KMR05], it is harder to derive a view schema for our views than for the views of Fan et al. Whereas several assumptions in the model of Fan et al. allow to derive easily a DTD to represent the set of all view documents, in our model the set of all view documents needs not even be regular. Our contribution regarding schema languages is twofold: on the one hand, we provide an efficient algorithm to test if a candidate view schema DTD satisfies the determinism constraint, and on the other hand, when the view schema is not regular, we provide regular approximations of the view schema, and study sensible restrictions that guarantee the regularity of the view schema.

Comparing policies in terms of determinacy and certain answers A major issue raised by views is the question of expressiveness: what information can be extracted from the view. In a security framework, we must check that the user cannot infer sensitive information from her set of authorized queries. We provide a few tools for comparing policies. Actually, we observe that some problems of policy comparison are related to the problem of deciding whether a query is determined by a particular view: given a query q on the source document, and a view V, can we answer q relying solely on $V(t)$ for all possible documents t? This problem, known under the denomination of "*determinacy*" finds applications beyond access control for the optimization of query evaluation with a cache (or with materialized views). We characterize the complexity of this problem in our setting(s).

The view update problem While the definition of an access control policy is relatively straightforward for read-only queries, "write" queries that modify the document (i.e., updates) highlight a new set of challenges. A well-known challenge is the view update problem: given an update u_v from the user (on the view), apply an update on the source document whose effect on the view is that of u_v. In the process of computing the update we will apply on the source document, i.e., the translation of u_v, we may have to choose between different possible translations. Possibly also, there may be no possible translation, when the policy forbids all write queries for instance. We first propose a solution to the view update problem when the document to be updated is fixed. Then we tackle the more general problem of translating *update functions*: if the user wishes to apply an update of the kind "delete all b nodes" whatever the document, we want to compute an update function that will delete all b nodes of a source document while preserving schema constraints.

Schemata with deterministic regular expressions We investigate in particular schemata (DTDs) satisfying the determinism requirement from the W3C standard (according to these standard, the regular expression used in a DTD or XML Schema should be deterministic for compatibility with SGML). We investigate to what extent our approximations remain "feasible" when the schema should be a DTD with deterministic content models, but also propose a linear-time algorithm to test if a regular expression is deterministic, whereas existing algorithms had quadratic complexity when the size of the alphabet is not bounded.

Publications

The results in this dissertation have been (partially) published in four conference papers.

- Our model of security views was first introduced in [GSC+09]. This paper also presented our schema approximations. However, the paper only considers views defined by DTDs with $\mathcal{X}Reg$, and covers a small fraction of our results on determinacy or schema approximations. A larger subset of our policy comparison results is currently under revision for a special issue of Information and Computation.

- Our first contribution to the view update problem for XML views appeared in [SBG10]. This paper tackles the view update problem when both the original document and the view update are given as input.

- While the previous paper focused on the optimality of the view update's translation, [BGT$^+$11] instead investigates translatability of update functions in a more general setting, when the original document is not fixed by the input.

- The algorithms to test determinism of regular expression have been published in [GMS12], together with algorithms to evaluate deterministic regular expressions .

1.4. Organization of the Manuscript

Chapter 3 presents definitions and general results, mostly about visibly pushdown automata. We essentially survey and fine tune existing algorithms with low-degree polynomial for problems such as membership or emptiness. Chapter 4 introduces our model of security views, with the corresponding algorithm to rewrite queries to bypass view materialization. We then investigate techniques for comparing different views in terms of determinacy. Chapter 5 is devoted to the view update problem for upward-closed security views, in different settings. Chapter 6 proposes several approximations for the schema, and explores in particular the effect of deterministic content models when the approximation is a DTD.

2. State of the art

The first part of this chapter is devoted to the main access control models for XML documents in the literature. The second section investigates the questions that arise when querying data through views. We put emphasis on two questions of immediate relevance for non-materialized views, namely which languages allow to evaluate directly on the original document a query that has been expressed on the view, and conversely when is it possible to answer a query that has been expressed on the original document, using only the view? The third section surveys the techniques adopted in databases to handle the combination of updates and views. It essentially focuses on techniques to update data through the view, and techniques to maintain or verify properties of the views when data can be updated. Finally, the last section mentions solutions to provide simple schemata for XML documents, discussing approximations of XML schemata and context-free grammars. Statistics on the structure of real-life XML documents conclude this overview with some insight on the relevance and limitations of the techniques in this dissertation, regarding the average depth of the documents and availability of a schema.

2.1. Access control specification for XML

Numerous formalisms have been considered to specify security views: annotations of the source document, schema annotations, sets of rules identifying target objects with XPath, etc. The eXtensible Access Control Markup Language [Mos05] standard spearheaded by the OASIS consortium defines both a language to specify security policies, and a language to submit or answer authorization requests. Similarly to many models considering both write and read operations, a policy is essentially specified as a set of rules, together with a rule combination algorithm. A policy may actually involves other elements, such as obligations defining actions that must be fulfilled in conjunction with the authorization decision, such as sending a notification email. Each rule comprises the target of the rule, and its effect (with possibly an additional condition on the applicability of the rule). The effect of a rule indicates whether the action is authorized or rejected; it may only take values “Permit” or “Deny”. The target of a rule consists of a resource, a subject, an action, and an environment: the subject defines the entity that wishes to perform the action over the resource. The action describes the list of actions requested on the resource. Typical actions attributes are `read` or `update`...

Environment attributes can be used to specify additional information such as time and date of the request. Attributes can be designated by XPath rules evaluated on the context. For controlling access over an XML document, the request context will typically include the XML document itself, so that subject and resource attributes can be obtained from the evaluation of the corresponding XPath queries over the document. In a nuthsell, a rule may be applicable if the attributes of the target (subject, action, resources, environment) are matched in the request context. Applicable rules are then combined at the policy decision point, and the decision is transmitted to the policy enforcement point together with obligations.

XACML is a very flexible and expressive language, but is very verbose and possibly difficult to master. Therefore Abassi et al. [AJREF10] investigate tree automata techniques to derive a representation of an XACML policy by an annotated schema. They thus provide a translation from a fragment of XACML into security views. Among the other access control policy languages, XACL [KH00] is credited with the introduction of provisional authorization in XML access control. Provisional authorizations are actions that have to be performed for the action to be executed. This embraces actions such as logging in, signing a term and conditions statement, etc. XACML also supports provisional authorizations, along with many other features from XACL. Kudo and Qi [KQ07] introduce and compare three implementation schemes for these two models.

The question of structure: what should be "hidden". Many access control models for XML consider the nodes of the document as the smallest unit of information. Models may differ in the way they consider attributes and other data (text) values but we abusively consider attributes as nodes. For an overwhelming majority of these models, security specification is only about granting or denying access to each node since the information is carried by nodes individually rather than by relations between nodes. A security policy is thus characterized by which nodes it allows to access.

Already this simple model raises a few questions in tree-structured documents: in the relational model (and as long as no there is no key preservation constraint) there is no ambiguity about which table results from the deletion of a particular tuple, whereas the deletion of internal nodes in a tree (locally) modifies the structure of the tree. Some security policy may disclose a node while hiding some of its ancestors. There are several approaches to tackle this question, although papers are not always very explicit about which one they embrace. Many proposals simply rule out the possibility of disclosing a node with invisible descendants: the *downward denial consistency* enforces the deletion of the whole subtree below a node that is hidden. This is the assumption in [MTKH03, DFGM08, LLLL11] for instance. Other frameworks define more expressive policies, that allow the disclosure of a node while anonymizing the label of its ancestor into a “dummy” label. The model by Fan [FCG04] could be ranked among those, modulo technical details (in some cases, nodes are simply deleted and their visible descendants are adopted as in the approach discussed hereunder). To further prevent the disclosure of sensitive information, other models fully delete invisible nodes from the view document, and a visible node is adopted by its closest visible ancestor. This approach has been adopted in [KMR09], for instance, and is investigated in the present dissertation, although we also allow the policy to specify anonymization instead of deletion for invisible nodes, as a special case of relabeling. We will additionally discuss the impact of downward denial consistency on the complexity of our algorithms. The chapter 5 about updates reasoning does not consider such general views and allows only the anonymization of internal nodes and the deletion of whole subtrees. Those approaches are certainly no panacea for the problem of protecting structural relationships, as information about the hidden parts could still be inferred, from the sibling order for instance. We briefly discuss at the end of this section some other approaches to protect structural relationships, but will not explore those directions of research: to the scope of this dissertation belong neither views restructuring the document nor statistical approaches.

The question of granularity: concise yet precise specifications for what should be “hidden”? Annotating each node of the document with its authorization status is not practical, so several methods have been proposed to provide concise specification of the policy: the access control rules are generally specified via the annotation of a schema [WSL$^+$07, FCG04] and via XPath queries [MTKH06, DdVPS02, FCG04]. An overwhelming majority of those models support propagation mechanisms to facilitate the specification of the policy. Typically, the accessibility of some nodes will be propagated to its descendants as long as it is not overridden. Depending on the models, this may be the default behaviour or it may have to be explicitly expressed in the access control rules.

In the case of access control models that rely on runtime evaluation of the policy, the efficiency of the policy representation has received particular attention. Yu et al. [YSLJ04] propose to use the structural locality of accessibility in order to build a space efficient representation of the accessibility map, i.e., the function mapping each node to its accessibility status. They essentially observe that it is not necessary to tag each node with its accessibility status: they optimize the number of tagged nodes and enhance those nodes with propagation tags so that the accessibility of each node can be deduced from its closest tagged ancestor and descendants.

Zhang et al. [ZZSZ07] refine this approach by using not only the structural locality of the rights, but also correlations between the rights of several users to further compress the accessibility map, where Yu et al. would store one accessibility map per role.

2.2. Access control models for XML

In this section we survey different access control models for XML. We first mention approaches that do not rely on the construction of a view (whether materialized or virtual) but allow the user to directly query the source document instead. We then investigate view-based access control models, in which the user queries the database through the view, and should typically ignore up to the existence of hidden nodes. The last two paragraphs present additional issues that we do not consider in this dissertation, namely how to specify access control models that can both protect relations instead of individual nodes, and handle data distributed on the web.

Access control without views Different techniques have been investigated to efficiently specify and enforce access control in the absence of views. The absence of a view makes it necessary to enforce the policy at runtime, i.e., together with query evaluation. In the security view framework, the user can only access the document through the view, and the schema provided to the user (if any) is that of the view. *Authorization transparent models*, on the opposite, allow the user to directly query the database. One potential asset of authorization transparent models according to Rivzi [RMSR04] is that they may save some development cost due to the necessity for the application programmers to code one interface per authorization view in view-based models.

Such authorization-transparent models have been investigated among others by [Mot89, RS01, RMSR04] in the relational setting, and [KMM06] for XML. Rivzi [RMSR04] distinguishes two different classes of models: in the *Truman models*, the query from the user is "rewritten" into a query that only accesses the information authorized by the view, while in the *non Truman models* the system checks if the query is "valid" and executes the query without modification if so, or else rejects the query and notifies the user. In the Truman model, when query answers exceed the authorized data, the rewriting process allows to return those of the answers that are within the authorization views. Even if the rewriting process is transparent, the answers delivered may not match the user's expectations, as illustrated in [RMSR04]. Non-Truman models, on the other hand, raise the question of how we should define validity. Validity is often expressed in terms of rewriting queries using views. Our own work in Section 4.2 also addresses this query rewriting issue, though from a different perspective. A contrario, this problem of query rewriting is radically different from what is called query rewriting in the *Truman models* (essentially, an intersection of the query with the set of accessible nodes) and from what is called query rewriting in the non-authorization transparent models for non materialized views (essentially, a composition of the query with the view).

In the Truman model, the simplest solution is to evaluate the query and verify during evaluation that each node accessed is accessible. In order to enforce access control at runtime with node filtering, one must then take care that not only the nodes returned but also the nodes examined by the query, inside XPath qualifiers for instance, are authorized to the user. This solution which might prove expensive when the policy is specified with queries or propagation rules that make it hard to establish accessibility of the node. Most proposals are about improving the efficiency of this approach, using indexes, static analysis or query rewriting.

Runtime evaluation of policies may be too expensive as it induces an overhead to the evaluation of each query. Murata et al. [MTKH06] lighten the burden on runtime analysis with a preliminary static analysis of the user's query. Static analysis techniques allow to distinguish queries that are guaranteed to be safe from those that cannot be answered and those whose correctness depend on the data. In this approach the user's query is preprocessed independently from the actual document. They essentially consider policies defined by a set of positive rules using only downward axes of XPath, which can easily be converted into regular expressions over paths by making abstraction of the filters. In the conference version of [MTKH06], denial downward consistency is explicitly assumed. However in both journal and conference versions, the syntax of the policy languages distinguishes rules that deny the access to a node and all its descendants (labeled $-R$) from rules that apply only at the node and are not propagated downward (labeled $-r$). This distinction seems rather useless under the downward denial consistency assumption, but was possibly introduced for symmetry with positive rules that grant access to the nodes. In the journal version furthermore, the authors also consider the possibility to allow a hidden node to have visible descendants, in which case the hidden ancestor node is anonymized. Let us quickly describe the static analysis algorithm. The static analysis algorithm first builds a regular expression (or NFA) from the query and two NFAs from the schema and the policy rules. When the query or policy rules contain filters, the analysis resorts to approximations by underestimating and overestimating the automata. The filters are abstracted as false in underestimation automata, and as true in overestimation automata. The static analysis determines from those automata if the query is unsatisfiable, or if it can be safely evaluated, or if additional filters need to be evaluated at runtime. Query rewriting is a popular approach [LLLL11, BP10] to avoid those runtime verifications, or more exactly, to integrate them into the query to be processed. The possibly unsafe query is rewritten into an equivalent query returning only the authorized nodes.

View based models for access control In their influential 2002 article, Damiani et al [DdVPS02]. develop a model based on materialized security views. The view of a document is obtained by pruning from the document every node that is not visible according to the policy, except for those that have visible descendants. For those nodes that should be hidden but have visible descendants, the authors decide to fully disclose the node (but would hide, for instance, its attributes): "to preserve the structure of the document, the portion of the document visible to the requester will also include start and end tags of elements with a negative or undefined label that have a descendant with a positive label." This statement is generally interpreted as the disclosure of the node label, which would allow to conflate this approach with downward denial consistency, even if the framework of Damiani et al. could also be adapted into anonymizing the ancestor. After deletion of the nodes, the resulting document may be invalid w.r.t. the schema, though. Therefore, the schema is loosened: every element from the schema is made optional. They consider authorizations specified either at the schema level or at the instance level, i.e., dealing with a specific node of a specific document. The authors enhance their model with *write* authorizations similar to the *read* authorizations. Those authorization rules specify the three privileges *insert*, *delete*, and *update*. [1] Processing *write* operations requires special care. For instance, the authors observe that, when inserting a node, the visibility of the inserted node must be checked in the resulting document. If the node is not visible in the new document the insertion operation is rejected. Furthermore, the authors observe that compliance of the new document with the schema cannot be taken for granted in general. The *write* operation is rejected if the new document is not valid with respect to the schema. The authors, however, do not study the side-effect issue: due to the high expressiveness of their authorization specification language, operations on an element may affect the visibility of another element.

Stoica and Farkas [SF02] advocate the use of views as a technique that guarantees better data availability and eludes illegal inference channels observed in previous approaches. They introduce a model of security views based on DTD annotation. Contrary to the models of Fan et al. [FCG04, FGJK07] and Kuper et al. [KMR05, KMR09], the annotation of the DTD is based on the tag of each individual element: the parent's tag is not taken into account. The specification of the security view are not the main focus of the paper, however: the paper addresses the problem of preserving association between nodes in a multilevel security policy. For this purpose, the authors introduce minimum semantics conflict graphs, which specify the associations (between pairs of nodes) that have to be preserved in the view. The authors show how security views can be constructed from those graphs and the DTD

[1] Since we do not consider attributes but only element labels, we use the term of *relabeling* or *renaming* in this dissertation, and save the word update for the document level

annotation.

Kuper et al. [KMR05] use a model closely related to the framework of Fan for non-materialized security views, except they propose to materialize the view instead of computing a new query for the composition of the view with the user's query. This clearly simplifies the task of evaluating the query, but on the other hand exposes the model to the drawbacks of materialized views underscored in [FCG04] in presence of multiple views and/or frequent updates. This can be remedied if the views are materialized on the fly as is also suggested in [KMR09], but this last solution may entail heavier query evaluation costs at runtime. The extended version [KMR09] discusses several materialization strategies. This paper also compares how different view models including [FCG04], [KMR05], [DdVPS02] handle hidden nodes.

Query rewriting for non-materialized security views Fan et al. [FCG04] introduced a first model of non-materialized security views that uses query rewriting. They carefully motivate their framework by comparisons to other approaches. We detail the main features of the model on page 128. In this model, the views may hide internal nodes by anonymizing them.

Although the original paper by Kuper et al. [FCG04] develops a framework for materialized views, the same setting was afterwards considered for non-materialized views that use query rewriting [Ras06] with a corresponding implementation [Ras07]. One of the distinctive features in the Kuper et al. adaptation of Fan et al.'s framework is the semantics for internal hidden nodes: those are deleted by the view in [KMR05], which triggers the adoption of each visible node by its closest visible ancestor, whereas in [FCG04, FGJK07] the nodes are typically anonymized. What is more, the semantics of filters in annotations differs in the two models. Another divergence lies in the DTDs considered: Kuper et al. consider general DTDs, whereas Fan assumes normalized DTDs in which the production of a node is either a disjunction or a sequence, but cannot be an arbitrary regular expression. This difference partly accounts for the choice of dummy nodes in Fan's models: dummy nodes are inserted in order to preserve the structure of the DTD when internal nodes are hidden.

The XPath fragments used in the models of Fan et al. [FCG04, FGJK07] only include the downward axes, whereas the fragments used in the model of Kuper et al. [Ras07, KMR09] also includes upward axes. The views in [FCG04, Ras07, KMR09] are non-recursive, but Fan et al. [FGJK07] consider a model allowing recursive queries and views, based on their original model [FCG04] regarding the specification of the policy: the main contribution of [FGJK07] is an algorithm to efficiently evaluate (downward) Regular XPath queries over non-materialized recursive views, as discussed on page 128. All those models have been implemented [FCG04, FGJK06, FGJK07, Ras07, KMR09], and the model of Fan et al. is protected by US

Patent 7433870.

Damiani et al. [DFGM08] propose another framework to enforce access control by query rewriting over non-materialized security views. The policy is defined by an annotation of a schema D (defined in XML Schema), in a way similar to Fan et al. The construction of the view schema, however, is quite simple because Damiani et al. assume denial downward consistency. Therefore, a rough approximation of the view schema is obtained by a simple loosening of the schema: the elements from D whose visibility is conditional are made optional in the view schema and elements from D which are not visible are simply removed in the view schema together with all their descendants. A DFA is also constructed to represent the annotated schema. When the user sends a query on the view with the assistance of the view schema, this automaton is used to rewrite the user's XPath query into a corresponding XPath query over the source document. The framework also supports the three write privileges *delete*, *insert* and *update*. As for read privileges, authorizations to perform those operations are specified by annotation of the schema. Although denial downward consistency is assumed, and XPath expressions are restricted to the downward fragment of XPath, yet one cannot always prevent the updates from raising side-effects on the source. For instance, deleting a node implies the deletion of all its descendants, some of which might be invisible nodes. The authors consider this to be a case of confidentiality versus integrity, and opt in favour of confidentiality: rather than rejecting the deletion of nodes with invisible descendants, they prefer to delete its invisible descendants as well.

Protecting structural relationships The question of ancestor visibility for hidden nodes is a prominent example for the difficulty of hiding structural relationships in XML. This protection of structural relationship has also received some attention from the community. Finance et al. [FMP05] propose to extend the rule based access control models for XML with relationship authorizations. Beyond the usual rules specifying the visibility of nodes, the database administrator specifies explicitly the sibling and ancestor relationships that must be hidden with relationship authorizations. The authors propose methods based on cloning and shuffling to prevent inference of information about sensitive relationships.

Inspired by the Chinese wall model [BN89], Cuppens et al. [CCBS05] propose an alternative approach to extend existing XML access control models with a mechanism for protecting relations. In this approach, the access control policy defines blocks of rules. The relationships between nodes selected in a same block are displayed in the view, but not the relationships between nodes from distinct blocks. The view is computed in two phases: the selected nodes are first computed independently for each block, then the resulting view trees (one per block) are merged. The merge process can fail if the policy is not well designed. In this model, as in [FMP05], some nodes from the original document can be duplicated in the view into anonymous copies. This model also tackles the risk of inferences from the sibling ordering by proposing a shuffle operation that randomly reorders the children of a node.

Security views restructuring the document could be an attracting alternative to the fragmentation of the policy into blocks. Creating new nodes, as well as copying, deleting or moving existing nodes are primitives proposed by the security specification language of Mohan et al. [MKSW06]. This proposal adheres to the query rewriting approach: the user formulates XPath queries on the basis of a view schema, and these queries are rewritten into XQuery programs evaluated over the source document. This expressive framework provides powerful tools for hiding structural relationships, but this may result in complex transformations, even though a non-recursive framework is assumed. These have been considered unfit for large-scale security specifications [DFGM08]. Furthermore, this framework assumes a non-recursive schema.

In their authorization-transparent model (which does not feature restructuration of the document), Kanza et al. [KMM06] also study the protection of structural relationships. Their approach is radically different from the ones mentioned above because of the authorization transparent model, because of a different formalism to specify the policy, and because the problems they tackle assume a fixed document as input. The policy is specified by rules of the form `for` p_1 `exclude` p_2, i.e., pairs of XPath formulae p_1, p_2. A rule of this form specifies that in the document t the relationships between the nodes in $[\![p_1]\!]_t$ and those in $[\![p_1/p_2]\!]_t$ must be concealed from the user. Actually the relationships are the child and descendant relationships between the nodes of those two sets, because the paper only considers the downward axes of XPath, although the authors mention the possibility to apply the same technique for general XPath queries. The authors define a "validity" property for queries which essentially guarantees that a valid query Q returns the same result for any pair of trees that differ only by their hidden relationships. The authors also study the coherence of a set of rules, and present an algorithm to check what level of security is guaranteed by a set of rules. In the spirit of k-anonymization, they define the relationships between two sets of nodes A and B to be k-concealed if for every element in B there are k elements $a_1, \ldots, a_k$ in A, and k documents indistinguishable by valid queries, such that in the i^{th} document b is a descendant of a_i but of no other $a_j (j \neq i)$. The authors give an algorithm that computes for a given document and for each rule of the policy the maximal value of k for which the relations between the sets $[\![p_1]\!]_t$ and $[\![p_1/p_2]\!]_t$ are k-concealed.

Extensions of the access control framework With the boost of distributed data management due to social networks and cloud services, the main privacy and security issues may now be raised by distributed data model. The security framework we consider handles a single document, stored as a whole on a single machine, and as such, this centralized model does not address the specific issues occurring in the context of distributed data. Several models have been proposed for reasoning about shared data in heterogeneous environments. The *Active XML* language, for instance, defines documents containing calls to Web services, together with a peer-to-peer architecture to manage those services [ABM08]. While *Active XML* is specific to XML documents, Abiteboul et al. [ABGA11] also propose the *Webdamlog* language, with the view to establish formal foundations for distributed data management. Instead of the calls to Web services from *Active XML*, *Webdamlog* uses Datalog-like rules to specify intentional data. The rules can be used for *delegation* (installing rules at another peer), for materializing a view of the data, for the sending of messages to other peers, or for deriving locally intentional facts, whereas facts "capture both local tuples and messages between peers." *Webdam Exchange* [GAP11] extends this *Webdamlog* language with policies controlling the access and the distribution of the data. In particular, the notion of *principal* allows a finer granularity of the access control than the physical peers used to localize the data in *Webdamlog*. Access control lists allow to specify `read` and `write` authorizations, as well as localization rights for deciding where data can be stored and found. Credentials (e.g., cryptographic keys) are also supported. *Webdam Exchange* thus provides a unified model to tackle the management of distributed data in a Web scenario.

Focusing on access control issues, Capitani et al. [dVFJ+10] address in a non-XML framework the questions raised by data outsourcing, where the data is stored on "honest but curious" external servers. Her model relies on encryption of the data since an honest-but-curious server is trusted to keep the data available, but should not be given access to the information, therefore the server cannot directly enforce the policy. Foresti supports policy updates without re-encrypting the data by using a two-layered encryption, the first layer protecting the privacy of the data, and the second layer enforcing the current version of the policy. Fragmentation of the data can be used to protect associations of sensitive information, possibly combined with encryption, as in [ABG+05, CdVF+10].

2.3. Queries on views

This dissertation investigates several questions about queries and views. In the non-materialized setting one must compute a query for the composition of the view with the user's query. We survey here related work on query composition. We then present the related work on determinacy and query rewriting because we use these notion for the comparison of policies. Last, and since views can be seen as a particular case of incomplete information, we mention some results on querying in presence of incomplete information.

Query composition In the problem of view and query composition, a view Q_v and a query Q over the view are given as input, and one must find a query Q'' over the source such that Q' is equivalent to the composition of Q and Q_v. Formally, one has to compute a query Q' such that for every document t, $Q'(t) = Q(\mathcal{V}iew\,(Q_v, t))$. We observe that such a query Q' always exists if we do not constrain its language.

Benedikt and Fundulaki [BF05] define *subtree queries* and study their closure under composition. A subtree query can be seen as an upward-closed view, with the additional requirement that leaves of the view trees must be leaves also in the original tree: for every tree t and subtree query Q, $Q(t)$ returns a document whose root-to-leaf paths are root-to-leaf paths of the original document. The definition of subtree queries is based on the XPath syntax and more specifically, of the fragment XPath($\Downarrow, \Downarrow^*, \Uparrow, \Uparrow^*, \cup, [\,], \wedge, \vee, \neg$). The *evaluation under subtree semantics* of an XPath query $Q_{\mathcal{X}}$ over tree t returns the subtree[2] $\lfloor Q_{\mathcal{X}} \rfloor(t)$ of t obtained by keeping every node selected by $Q_{\mathcal{X}}$ under the usual semantics ($Q_{\mathcal{X}}(t)$), plus all their descendants and ancestors. The authors study the closure under composition of subtree queries, i.e., establish which fragments F guarantee, for every pair of queries $Q_1, Q_2 \in F$, the existence of a query $Q \in F$ satisfying for every t, $\lfloor Q \rfloor(t) = \lfloor Q_1 \rfloor(\lfloor Q_2 \rfloor(t))$. Among others, their results show that tree patterns are not closed under composition, but union of tree patterns are. Computing the composition for union of tree patterns involves an exponential blowup, but for fragments with upward axes, the authors obtain simpler polynomial algorithms.

Vercammen et al. [VHP06] study the closure under composition of a larger XPath fragment under the more traditional semantics: the view needs not even be upward closed, and the view tree corresponding to query Q_v is built from the set of nodes selected by query q_v, each node being adopted by its closest visible ancestor. They distinguish fragments $\mathcal{C}$ of XPath that guarantee for every $Q_v, Q \in \mathcal{C}$ the existence of a query $Q'' \in \mathcal{C}$ equivalent to the composition of view Q_v and query Q. The authors study several fragments of NavXPath extended with path intersection and path complementation operators, which makes this fragment closer to XPath 2.0 than XPath 1.0. Some of these fragments are not closed under composition: essentially those are the fragments excluding path complementation but including recursive axes, or sibling axes, or union. The remaining fragments studied in the paper are closed under composition. What is more, query Q'' can be computed from Q_v and Q in time $O(|Q_v| \times |Q|)$ or $O(|Q_v|^2 \times |Q|)$, depending on the fragment.

Fan et al. [FCG04, FGJK07] study the problem of view and query composition for their security view frameworks. They specify the view by an-

[2] In their semantics, evaluation of a subtree query returns a tree, not only a set of nodes. As usual, the structure of the view tree is inherited from the original tree

notated DTDs, with annotations defined in the same XPath fragment that is used for querying the view: downwardXPath in [FCG04], and downward Regular XPath in [FGJK07]. In general, XPath($\Downarrow, \Downarrow^*, \cup, [\ ], \wedge, \neg$) is not closed under view composition for recursive DTDs [FGJK07], but it is when the view is non-recursive [FCG04]. This is because axis $\Downarrow^*$ does not allow any control over the nodes it "skips", unlike the Kleene star operator on path expressions: Regular XPath($\Downarrow, \Downarrow^*, \cup, [\ ], \wedge, \neg$) is closed under composition, even for recursive views. Computing the composition, however, involves an exponential blowup in the size of the formula, and the same exponential blowup is incurred by XPath($\Downarrow, \Downarrow^*, \cup, [\ ], \wedge, \neg$) over non-recursive views. Fan et al. introduce an automaton formalism to circumvent this exponential blowup [FGJK07]. This formalism is closely related to alternating word automata, but alternation is used for branching over possibly different paths in the tree. We briefly compare our view and query composition algorithm with the one of Fan et al. on page 128.

Compared to the problem of view and query composition, the problem of answering queries using views reverses in some sense the role of Q' and Q: its input consists of a view Q_v and a query Q' over the source, and one would like to answer query Q' by relying only on the view Q_v. In other words, one has to write a query Q that takes as input the view of the document by Q_v (but not the document itself), and returns the answers of Q' over the document. Contrary to the composition problem, the view Q_v may provide insufficient information to answer query Q', so that allowing unlimited computational power to Q does not guarantee a solution to the problem. The connection between the two problems has already been observed in, e.g., [FGJK07].

Determinacy and query rewriting The problem of answering (or rewriting) queries using views comes in two flavours: one should first decide if the view provides enough information to determine the answer of the query for every document. This is known as the *determinacy* problem. Knowing that view V determines the answer to query Q, however, is of little help in practice if we do not know how to compute the answer, or if the computational power demanded is unreasonable. In the problem known as *query rewriting*, one has to compute in some particular query language a reformulation of query Q in terms of a query using view V. In the usual formulation of those two problems, V consists of a set of views and the document is an arbitrary structure.

Determinacy has been investigated in the middle of the 20$^{\text{th}}$ as a problem of implicit definability in logics. More recently, these problems of rewriting queries using a given set of queries have received considerable attention from the database community, for all sorts of view and query languages, including *MSO* and *FO*, conjunctive queries, and fragments of those languages [Mar07, NSV10, Pas11, Hal01], as well as SQL queries with aggregates [CNS99, AC11] and regular path queries [CGLV02, CGLV07]. Halevy [Hal01] surveys the different contexts in which the problem has been considered, from query optimization and database design to data integration and data warehousing. He also establishes the state of the art at the end of the 90's in the relational setting, and surveys the different algorithms with a particular focus on their application domain. Several different settings are distinguished for the query rewriting problem. Under the *exact* view assumption, the view document is guaranteed to consist in the set of all tuples selected by the view query, whereas under the *sound* view assumption the view only returns a subset of the tuples it selects. We only consider the exact view assumption in this dissertation. Similarly, we only consider *exact rewritings*, but in other contexts people have investigated *maximally contained rewritings*, namely greatest under-approximations of the query Q using the view V. In the present work, we came upon the determinacy and query rewriting problems when we tried to compare views. The tree structure of the document and our choice of query language makes it relatively easy to obtain some results in our framework. In contrast, those problems appear to be quite challenging for conjunctive queries(CQ) over relational databases. We first briefly survey the state of the art regarding graphs and relational databases, yet for a broader state of the art we refer the reader to [Hal01] for the motivations of query rewriting using views, to [NSV10] for a general overview, or to [Mar07] for a historical perspective.

The decidability of determinacy for CQ queries and views is still open. However, it is known that when a set of CQ views V determines a CQ query Q, one cannot always find a CQ rewriting of Q in terms of V [NSV10]. Nash et al. [NSV10] investigate the question of completeness of a rewriting language as well as the decidability of determinacy for view and query languages ranging from $FO$to CQ. Given a view language $\mathcal{V}$ and query language $\mathcal{Q}$, a language $\mathcal{R}$ is *complete* for $\mathcal{V}$-to-$\mathcal{Q}$ rewritings if for every set of views V in $\mathcal{V}$ determining some query Q in $\mathcal{Q}$, one can rewrite query Q in terms of V using some R in $\mathcal{R}$. Every language complete for FO-to-FO rewriting is Turing-complete in the sense that it contains all computable functions, and determinacy is undecidable for FO views and queries (it is even undecidable for unions of conjunctive queries) [NSV10]. The authors also show $\exists SO \cap \forall SO$ to be complete for CQ-to-CQ rewritings, and similarly for rewritings involving conjunctive queries with difference, or union of conjunctive queries instead of CQ. Moreover, they also study fragments of CQ for which not only determinacy is decidable but also CQ form a complete language for rewriting. These last results are further extended by Pasaila [Pas11]. Afrati [Afr11] also pursued the study of completeness for rewriting for fragments of conjunctive queries. She relates determinacy to conjunctive query equivalence in a particular setting with a single view, and she also introduces the notion of a language being *almost* complete for rewriting. Zheng and Chen [ZC11] investigate a different restriction of conjunctive queries: they show that determinacy is decidable in quadratic time when all relations have arity one, and in this case CQ is complete for CQ-to-CQ rewriting.

Fan et al. [FGZ12] also investigate determinacy, under the name of *invertibility*, for views and queries defined in CQ, Datalog and FO. Quite similarly to our concerns in this dissertation, the authors do not study determinacy for its own sake, but as a criterion for preserving information. They sharpen the undecidability result from [NSV10], proving determinacy to be undecidable when the view language is Datalog and the query language is CQ, or vice-versa. They also establish the complexity of determinacy when the query language is CQ and the view language is one of the SP, PC or SC fragments of CQ (where SP denotes the conjunctive query using selections and projections and PC, the conjunctive queries using projections and Cartesian products). The complexity for determinacy ranges from PTime for PC to NP-complete for SP ans SC, and drops to PTime for SP when the query Q is a minimal CQ. Finally, the authors introduce the *query preservation problem*, closely related to determinacy, and establish its complexity for the aforementioned query languages.

We so far discussed relational databases, but query rewriting techniques have also profited XML databases: the major database vendors have already developed XML-specific techniques for optimizing queries with materialized views[3] [KLM+04, GGH+09]. Furthermore, the other traditional applications for query rewriting such as data integration are possibly even more relevant in an XML framework.

Xu and Öszoyoglu [XÖ05] and Mandhani and Suciu [MS05] study the query rewriting problem when there is a single view defined by an XPath query. Mandhani and Suciu derive a sound but incomplete algorithm for query rewriting, when the view and query are defined by tree patterns. They assess experimentally the efficiency of their technique for optimizing a small fragment of XQuery in presence of materialized XPath views: their system maintains a cache containing the results of some XPath queries and, when a new query must be evaluated, it tests for each view in the cache if the query can be rewritten in terms of that view. Xu and Öszoyoglu prove that deciding the existence of a query rewriting is co-NP-hard for tree patterns. The authors also prove query rewriting to be in PTime for the fragments of tree patterns that allow child axis but rule out one of the other features: descendant, wildcard labels or branching. They also claim an upper bound, but this upper bound is based on a paper since refuted, as observed by Afrati et al. [ACG+09]. Afrati et al. show the query rewriting problem to be decidable when the view and query are tree patterns. They also prove the problem to be co-NP-complete for a large fragment of tree patterns, and argue that their paper introduces radically new techniques for reasoning about tree patterns. Those works consider the query rewriting problem for a single view, but with the help of node

[3] materialized views are called *materialized query tables* in IBM DB2

identifiers, several views can be combined to answer queries. Several authors have therefore undertaken the study of query rewriting problems in presence of multiple views [BÖB+04, ABMP07, TYÖ+08, CDO08, MKVZ11]. Balmin et al. [BÖB+04] introduce an algorithm to rewrite XPath expressions using multiple materialized XPath views. The XPath fragment considered for the views and queries includes essentially the axes child, descendant and parent (but not ancestor nor any of the optional axes from XQuery's Full Axis Feature). The rewriting algorithm also handles comparison predicates. According to the authors, value-based comparison play a crucial role in query optimization owing to their high selectivity. The query rewriting algorithm comes with no completeness guarantee, but implementation of the system allows to assess the efficiency of the optimization framework. Arion et al. [ABMP07] study the containment for queries expressed in a "rich" tree pattern language, under Dataguides enhanced with integrity constraints. Dataguides [GW97] form a structural summary describing all paths that occur in the document, which incidentally bounds the depth of the document. The authors derive a sound and complete algorithm for query rewriting under Dataguide constraints, and report on practical performances of the ULoad prototype that implements these algorithms. Tang et al. [TYÖ+08] use a path decomposition of the tree patterns to filter out the views that cannot be used to produce a rewriting. They require an encoding of node identifiers that records the label of all ancestors of the node, and use this encoding to combine the views for the query rewriting. Cautis et al. [CDO08] show that containment already becomes intractable when wildcard-free tree patterns are extended with path intersection (in the spirit of XPath 2.0, but only at the top level: intersections cannot be "nested" and must occur at the "root" of the pattern), which implies intractability for query rewriting. Consequently the authors identify restricted settings that allow to rewrite their XPath queries using an intersection of views in polynomial time. The authors also prove that deciding if the intersection of patterns can be rewritten without using intersection (nor union) is co-NP-hard, but is in PTime under restrictions similar to those for the query rewriting. Manolescu et al. [MKVZ11] consider the query rewriting problem for XQuery views and query. They actually consider a fragment of XQuery corresponding to tree patterns enhanced with value joins: those tree patterns may return several output nodes, and a view may specify joins between nodes from different patterns. Their algorithm first filters out views that cannot contribute to the rewriting, and afterwards computes then optimizes a rewriting. The authors put great emphasis on the optimization of the rewriting: they obtain rewritings that use no redundant views. The rewriting algorithm may exploit but does not require structural (node) identifiers. Finally, the authors compare experimentally the performances of different strategies for computing the rewriting. Cautis et al. [CDOV11] study a framework in which the set of views is very large (and possibly infinite), precluding explicit enumeration of the views. The views are therefore defined implicitly: a grammar-like formalism specifies the family of xpath views which are available for the rewriting.

Bohannon et al. [BFFN05] study related notions though in a very different framework. They consider a function that maps each document satisfying one DTD D_1 to a document satisfying another DTD D_2. Such a function σ is *invertible* if the original document can be recovered from the target document. Similarly, the function σ is *query preserving with respect to a query language* L if there is a computable function $F : L \to L$ such that for any $Q \in L$ and any document t satisfying D_1, $Q(t) = F(Q)(\sigma(t))$. In short, σ is query preserving w.r.t. L if every query from L can be rewritten as the composition of another query from L with σ. The paper considers schema mappings, whereas we consider views. What is more, we distinguish two settings depending on whether identifiers are taken into account or not, a distinction that is absent from [BFFN05]. Their definition of invertibility may be considered under both settings. We observe that when considering identifiers, $Q_1 \preccurlyeq_2 Q_2$ if and only if Q_2 is query-preserving w.r.t. $\mathit{Public}(Q_1)$. Also, Id $\preccurlyeq_3 V$ if and only if V is invertible in the sense of [BFFN05], where Id is the identity query, i.e., the view that hides no node. For this, we must consider invertibility without identifiers in our model: if we assume each node has a (unique and arbitrary) identifier, every query V that deletes an unbounded number of nodes would not be invertible due to the impossibility to recover the identifiers of the hidden nodes.

Incomplete information The questions raised by updates on the views and those raised by query answering using views mostly derive from the incompleteness of the information provided by the view. Incomplete information is arguably the norm rather than exception in databases. The theoretical foundations of incomplete databases date back thirty years [IL84]. Abiteboul et al. [AKG91] establish the complexity of several problems pertaining to incomplete databases. The field has received renewed interest since then, in parallel with the maturing of data exchange and data integration techniques.

Certain answers are allegedly a tool of choice when it comes to handling incompleteness. The *certain answers* of a query with respect to a partial information, are generally defined as the intersection over all possible documents of the answer set. The problem of computing the certain answers of a query given a view document has received much attention under the name of *query answering* and is also related to the problem of rewriting a query using views. The above definition for certain answers does not fit in the case of XQuery queries which return a single tree instead of sets of tuples for relational queries. David et al. [DLM10], in particular, remedy the shortcomings of the usual definition with a homomorphism-based definition of certain answers. For queries returning sets of nodes, Barcelo et al. analyse the complexity [BLPS09] of the standard questions related to incompleteness, such as consistency, membership and query answering. They focus on restrictions that lower the complexity of those problems, and observe in particular that schema information quickly makes the problems intractable.

Several specific models of incompleteness have been proposed for XML documents [ASV06, BLPS09, DLM10]. Libkin investigates a general data model subsuming relational and XML frameworks for incompleteness. This model uses a homomorphism-based ordering on the information conveyed by the incomplete database in order to measure the "degree of incompleteness" [Lib11]. Libkin shows that seemingly divergent definitions of certain answers in relational and XML settings share the same interpretation in terms of greatest lower bounds in the general model. He also studies the interpretation in this general model of data exchange solutions in both relational and XML settings. The paper additionally surveys the complexity of the standard questions related to incompleteness in this general model.

In the more specific view framework, Libkin and Sirangelo [LS10] present as a subsidiary result an algorithm for reasoning about certain answers disclosed by upward-closed views. Given an upward-closed view V and Boolean query Q, their algorithm computes the set of all view trees from which one cannot be certain that Q holds on the source document. We discuss their results further on page 161. Kopczyinski [Kop11] investigates another problem more distantly related to the results in the present dissertation, namely the consistency of the incomplete information. He assumes a tree model similar to [BLPS09], and uses tree automata as a schema model.

Beyond certain answers issues, the whole fields of *data exchange* and *data integration* study the problem of answering queries using views and are also more or less related to the view update problem. The formalisms, however, are different from ours, as the relations between source and target are generally expressed with dependencies of various kinds, and it is not clear how our problems could be formulated in these settings, so we will not expand about those.

2.4. Views and Policies in Presence of Updates

A peculiar characteristics of works on XML updates, illustrated by the survey in [Che08] for instance, is the miscellany of update languages considered: most papers in the field define their specific update language. Those languages are nevertheless quite similar, and generally consist in extensions of XQuery with update operations. This section briefly surveys the problems raised by the possibility to update the document, from the *view maintenance* and *view update* problems to the more general questions of reasoning about evolving documents, with an emphasis on access control and related issues.

View Maintenance The problem of maintaining materialized views has received much attention from the database community: in the relational setting (see the studies of Gupta et al. [GM95] for a general overview, or Koch [Koc10] for a more recent, but also more specific algebraic perspective on incremental evaluation), for graphs or semi-structured data [ZGM98, AMR$^+$98], and more recently for XML documents [FKSV08, BGMS11]. Gupta et al. [GM95] survey several approaches to the problem. In the lucky case, the update might leave the view unchanged: this motivates the study of query-update independence, which we briefly survey in the next paragraph. In general, however, updates will affect the views, but query-update independence can still be useful as a preliminary step. It might also be the case that knowledge of the initial view and of the update applied is sufficient to compute the updated view, without additional information on the source document. This motivates the self-maintainability approach: views that can be maintained using only the view and key constraints are called self-maintainable but in general views need not be self-maintainable. Finally, *incremental maintenance* of the view is one of the prevailing approaches when re-computing the view from scratch appears too expensive, and this technique is implemented in major commercial databases.

Several practically-oriented papers addressed the incremental evaluation of XPath queries. More recently, Bjorklund et al. [BGM10] investigated the worst-case complexity of the problem for several fragments of XPath, but they almost exclusively focus on Boolean queries, as we discuss below. Maintaining XQuery views is even more challenging a task, but several approaches already tackle the maintenance of XQuery views. Bjorklund et al. [BGM10] established the complexity of incremental evaluation for several XPath fragments, but they essentially consider a Boolean restriction of view maintenance, namely the problem of deciding if a Boolean XPath query is satisfied by the document after the update, which is therefore related to the problem of query-update independence, with the restriction that for incremental evaluation, the document to which the update is applied is available. An auxiliary data structure is maintained in addition to the document. The updates considered encompass all atomic operations, and the authors propose algorithms with time polynomial in the size of the query, and sublinear in the size of the document for several fragments of (navigational) XPath, using an auxiliary datastructure of size linear in the document and polynomial in the size of the query. As an adaptation of a similar result by Balmin et al. [BPV04], they also show that incremental evaluation is feasible for the whole NavXPath dialect in time sublinear in the document and exponential in the size of the query, using an auxiliary structure of size linear in the document and exponential in the query.

Foster et al. [FKSV08] propose a system for the incremental maintenance of XQuery views. The query (view) language they consider is a fragment of XQuery, which is translated into algebraic operators similar to those of the relational model, but with some additional operators to define the navigation over the tree. This fragment can express FLOWR blocks and the XPath navigation with downward axes, however it does not handle the recursive features of XQuery. Therefore the view is essentially obtained by combining the results obtained through the evaluation of a set of XPath queries. This combination may involve joins, reordering, etc. The updates considered are expressed as a set of atomic update operations, carrying a query from XQuery that should be evaluated at runtime over the tree in order to specify the data that must be inserted or deleted. The authors allow the system to maintain an auxiliary structure. They demonstrate the performance of their prototype over views from the XMark [SWK+02] benchmark.

Bonifati et al. propose another algebraic approach for the incremental maintenance of materialized XML views [BGMS11]. As in [FKSV08], the view and update are defined by XQuery and XQUF expressions. The fragments, however, differ slightly: the syntax for the view language is simpler but less expressive. It still allows one `for` loop and also uses the downward fragment of XPath. As for the updates, deletions are specified via an XPath formula returning the nodes to be deleted, whereas insertion uses a `for` iteration to specify the data that must be inserted, depending on the context. While the approach of Foster et al. [FKSV08] exploits the tree algebra of Galax, the approach of Bonifati et al. is designed to benefit from standard optimizations techniques from the database engine, such as structural joins, delta tables, and term pruning heuristics. The authors also use XMark for benchmarking.

Independence of Updates and Queries (or Views) An update u and a query Q are independent on a document t if the answer of query Q over t remains the same after update u is applied to t. More generally a class of updates U and a query Q are independent if for every source document and every update in U, the view of the document remains the same after application of the update. This problem of query-update independence has received much attention, and in particular in the XML framework [RS06, STP+06, BC09, BGM10, GI08]. Query-update independence has been investigated essentially for the maintenance of materialized views but also as a mean of detecting conflicting updates, among others. Query-update independence may also find applications in the constant complement approach, in order to check that a given update does not affect a certain query (the complement). Benedikt et al. [BC09] for instance make a short observation about such possible use of query update independence in an access control framework to verify the compliance of an update with a policy that forbids to alter some view.

Raghavachari and Shmueli [RS06] study the problem of detecting conflicting updates, which in their approach amounts to a problem of query-update independence. They study the complexity of determining exactly if an update and an XPath query are independent. This study does not restrict the queries to Boolean XPath, which leads to a distinction between different kinds of conflicts, depending on the semantics of query evaluation (namely, does a query return the set of selected nodes or the whole subtrees below those nodes) and the kind of updates considered. The authors show NP hardness of the problem for fragments of XPath($\Downarrow^*, \Downarrow, [\,], \wedge$), and provide a polynomial algorithm for the XPath fragment that does not use filters. Sawires et al. [STP$^+$06] implement algorithms to test query-update independence and self-maintainability for the same XPath fragment as [RS06], when the document and its view are loosely coupled, a framework which previous algorithms do not manage well according to the authors.

Benedikt et al. [BBFV05] also study query-update independence in order to optimize the ordering of update operations for an XQuery-inspired update language. Their independence criterion is undecidable, so they provide a sufficient condition in terms of satisfiability of XPath queries. They apply the resulting independence check to speedup the evaluation of queries without the snapshot semantics.

Gire and Idabal [GI08] study the independence of views and updates when both the view and the update are specified by regular tree patterns. A regular tree pattern is essentially a tree pattern of which the edges represent regular expressions. Gire and Idabal show the Pspace-hardness of independence in their framework, and propose a sufficient condition for independence that can be tested in polynomial time. Gire and Idabal also propose to define functional dependencies using the same regular tree pattern formalisms and obtain similar results for the problem of independence between the update and the functional dependencies, i.e., the problem of checking if an update affects the satisfaction of a functional dependency [GI10].

Benedikt and Cheney [BC09] develop a schema-based static analysis algorithm to decide if an XQuery query and an XQUF update are independent. The schema language corresponds to regular tree languages (EDTDs), and the query and update languages correspond to restricted versions of the XQuery and XQUF languages: their core language for XQuery allows `for` iterations, `let` bindings, conditions, as well as the standard XPath axes. The updating expressions can similarly use iterations and bindings via XQuery selection queries in order to define the nodes concerned with atomic updates. The authors observe that query-update independence (with or without a schema) is undecidable for the general XQuery and XQUF languages, but becomes decidable, though non-elementary for their restricted versions when the query considered is a Boolean query. The decidability proof works by reduction to the equivalence of first-order logic formulae: one for the query and one for the composition of the query with the update. The non-elementary hardness is obtained by a reduction from satisfiability in first-order logic over trees. In view of those negative results, Benedikt and Cheney propose a "best-effort" analysis that approximates the set of nodes "impacted" by the update and checks they are disjoint from the nodes "accessed" by the query. Their analysis is sound, but not complete as some queries and updates are not detected to be independent.

The same authors introduced the notion of destabilizers [BC10], which provides a different, schema-independent framework for query-update independence. A *destabilizer* for a given query is a finite representation of the set of all updates that could modify the result of the query. In this framework, an update expression u is independent from query Q if the set of updates that can be generated by u does not intersect the updates represented by the destabilizer of Q. For the authors' core XQuery and XQUF languages, however, computing a destabilizer is not feasible, so that the authors resort to an over-approximation of the destabilizer. The authors compare experimentally two approaches to verify that update u does not intersect the destabilizer: the direct one uses solvers for satisfiability in monadic second order logic over trees, and the second one reduces the problem to satisfiability modulo theories via the encoding of trees. In any case, the disjointness analysis can be carried out with existing solvers.

Ghelli et al. [GRS08] develop a path-based static analysis to compute over-approximations of the nodes accessed or modified by an XQuery update. The authors actually study an update language of their own, XQueryU, based on an extension of XQuery with update operations. In particular, they do not assume snapshot semantics for the updates, which means that updates may be applied in the course of the evaluation. The static analysis method developed by the authors provides a conservative approach to determine whether two XQueryU updates commute.

Bidoit-Tollu et al. [BTCU12] propose a different schema-based analysis of query-update independences for a similar fragment of XQuery. Their approach infers the possible chains of labels along the paths from root to nodes, which allows to detect query and update independences overlooked by the previous algorithms, thus improving the precision of the over-approximation.

Update anomalies In presence of a schema, some updates cannot be executed without side effects or without making the database incoherent. Let us a consider a traditional "student and courses" database where students always appear as a pair formed by their name together with some identifier: their student number, for instance. The database consists of a set of triples (student name, student id, course taken). A typical constraint will be that the same identifier should not be given to two different students. This redundancy introduces update anomalies: if the name of some student is updated for some but not all occurrences of the student, the database will not satisfy the constraint any more. Also, a student cannot be inserted in the database until it has registered for some course, and the deletion of the last course taken by the student makes the student disappear from the database. We observe that in general the schema and constraints such as functional dependencies may restrict which updates can be applied without side effects.

Normalization is a widespread approach to avoid update anomalies in the relational setting. The process of normalization can be traced back to the original presentation of the relational model by Codd [Cod70], and the success of normalization inspired similar normal form proposals for XML documents [AL05]. A well-designed schema should then follow some normal form such as 3NF or BCNF that eliminates or rather minimizes redundancy. A better design for the example above would be to store the pairs (student name, student id) and (student id, course taken) in two distinct tables. Normalization algorithms generally decompose a database into smaller tables but, depending on the constraints, normalization is not always feasible. Some schemata, for instance, do not admit dependency preserving BCNF decompositions. What is more, normalization of the source schema does not prevent data redundancy when computing the views, and thus does not necessarily prevent side effects for updates applied through a view (view updates) [Feg10].

The View Update Problem: Generalities Commercial databases such as Oracle, IBM DB2, MySQL and Microsoft SQL server nowadays support view updates. Some undesirable side effects can be avoided using by checking that tuples inserted in the view can really appear in it according to the view definition.

The *view update problem* is in some sense symmetric to the *view maintenance problem*: in the view update problem one tries to maintain the original document when its view undergoes updates from the user, whereas the view maintenance problem deals with the maintenance of the view when the original document undergoes updates. Of those two problems, the view update problem is certainly the more challenging due to the question of side effects and the necessity to decide between different propagations. Most contributions to the view update problem date back to the 80's, and in particular the formalization of the problem and its main solutions. Before we survey the view update problem, let us mention a workaround: one may in some settings expect the view-update to have no consequences beyond the modification of the corresponding view, if the updated data is to be accessible only from that view. In this case there is no need to modify the document: one can remember the update and apply it "on the fly" at query time, by composition of the view with the update and with the user's query for instance. This approach is investigated by Fan et al. in [FCB07] for XQUF updates, in the spirit of *hypothetical queries*. Actually the *transform queries* investigated in [FCB07] can be applied in a much broader context, as argued by the authors: even in the absence of updates, non-materialized views can be modeled as transform queries.

The *translation* u_s of a view-update u_v w.r.t. view V has to be *side-effect* free, i.e., the view obtained after performing the translation u_s on the original document t must correspond to the application of the view-update u_v on the initial view. More formally, the following equality must be satisfied: $V(u_s(t)) = u_v(V(t))$. Early works by Dayal and Bernstein [DB82] formalize the correctness of translations, and characterize conditions under which such translations exists in the relational setting. Keller [Kel85] devised additional criteria in view of choosing translations that do minimal changes to the document. He suggests to use those criteria to reduce the number of possible translations. The decision between the remaining possible translations should be taken through dialog at view definition time, to best take into account the semantics of the real-world database [Kel86]. More generally, different criteria for correctness of a translation have been considered, formalized through the notions of complement, of consistent views, or other order- or information-based approaches.

The *constant complement* approach has been defined by Bancilhon and Spyratos [BS81] as a criterion for choosing a meaningful propagation of the view-updates. Their idea consists in defining a notion of *complement*: the complement of a view mapping V is a function C such that for every document t, the pair $(V(t), C(t))$ formed on document t by the view and its complement uniquely determines t. Although the paper is in line with the relational setting, the definitions of view mapping and complement in [BS81] are general functions mapping a document to an arbitrary value. One can define minimal complements for every view mapping, but they are never unique except in trivial cases. Minimality is defined with respect to information content: a mapping (view) C_1 will be smaller than another C_2 if it distinguishes more documents: $C_2(d) = C_2(d') \implies C_1(d) = C_1(d')$. In the constant complement approach, a complement for the view is fixed, and the translation of any view-update is required to keep the complement constant. If C is a complement for view V, every view-update admits at most one propagation under constant complement C. The translations under constant complement also enjoy nice properties with respect to reversibility: Lechtenbörger [Lec03][4] proves that in some sense the updates that can be translated under constant complement are the updates which can be translated in a reversible manner when the set of authorized updates is "complete" i.e., when the authorized updates are closed under composition and allow to undo any update. Constant complement strategies have often been considered a gold standard for view update translation because they eliminate update anomalies [Heg04]. The trouble with the constant complement approach is twofold: one has to choose a complement for the view, and there are numerous sensible updates

[4] The proof of this property in [Lec03] seems slightly inaccurate, but the author wrote a patch in a subsequent note

which cannot be translated under constant complement.

Soon after the seminal paper of Bancilhon and Spyratos, Cosmadakis and Papadimitriou study the complexity of computing minimal constant complements in the context of relational databases [CP84]. They actually consider a very restricted setting, with a single relation R and views limited to projections of the document, with integrity constraints expressed as functional dependencies. The views and complements are therefore essentially defined as a subset of the attributes from R. Even for those trivial projective views, computing a minimal complement turns out to be NP-complete. The definition of minimality in this paper is not the aforementioned one, but denotes the projection with the fewest attributes. The authors also characterize complementarity for projective views, and provide an algorithm with cubic data complexity to decide for a given complement C if the insertion of a tuple can be translated under constant complement C. They also study the problem of deciding for a given tuple i if there exists a complement that would allow to translate the insertion of i under constant complement. Those results are then extended to deletions and replacement operations. More recently, Lechtenbörger and Vossen provide algorithms with polynomial complexity that compute "reasonably small" complements to sets of views belonging to several classes of relational queries, yet without considering arbitrary functional dependencies [LV03]. The complements obtained are even minimal in some cases such as views that do not use projection, and the authors argue why minimality might sometimes be irrelevant. The authors also relate previous works discussing applications of the constant complement approach in data warehousing for self-maintainability: a view that is not self maintainable can be made so by the adjunction of complementary information.

The constant complement approach has often appeared as too restrictive in the sense that many reasonable updates cannot be translated under constant complement [Kel87, GPZ88]. Gottlob et al. extend the results of Bancilhon and Spyratos with the definition of *consistent views*. In the terminology of Gottlob et al., a *(dynamic) view* consists of a view together with an update policy (a.k.a. translator), i.e., a function that maps every update to a translation. A dynamic view is consistent if any two equivalent sequences of view-updates have equivalent translations. When all update operations can be undone (as it is assumed in [BS81]), consistent views correspond to translators under constant complement. But without this cyclicity assumption, there exist consistent views allowing updates that cannot be translated under constant complement: in other words consistent views do not require from the updates that they keep the information of some complement constant. Instead, consistent views admit a complement as defined in [BS81], whose information is "decreased" by the application of any update for an appropriate order. In a nutshell: the "correct" translations for consistent views form a strict superset of the translations under constant complement. We refer the reader to the article from Gottlob et al. for the definitions. Moreover, choosing the complement and the corresponding order uniquely determines the update policy, as for the constant complement approach. Another interesting feature of the article is its high level of generality: similarly to the paper by Bancilhon and Spyratos, this model is not specific to the relational setting and handles arbitrary mappings for the definition of (static) views. Furthermore, they consider *update programs*, which means that any update's semantics is defined as a mapping from D to D where D denotes the set of all documents. The authors also establish a thorough state of the art for the view update problem at the end of the 80's.

Hegner further investigated the constant complement approach in a collection of papers and articles spanning over two decades. To quote but a few: [Heg90, Heg04, Heg08]. These results focus in particular on the possibility to choose between different complements. They establish the difficulty to find complements of a given view that define "distinct but reasonable" update policies. The author proves that constant complement update policies are unique and independent of the complement under several natural assumptions, based on the ordering of all possible documents [Heg04] or on information-based techniques specific to the relational model [Heg08]. Closer to the topic of this dissertation, Hegner defines additional constraints that should be verified in *closed views*, i.e., when the user should ignore the existence of a view, the view being presented as if it were nothing but a schema for the user's document. In particular, he defines the notion of *uniform updatability*. A view-update is *uniformly translatable* if it can be translated whatever is the current state of the original document [Heg90]. Similarly to the works of Bancilhon and Spyratos or Gottlob et al., though with yet a different formalism, the results of Hegner are mostly model-independent [Heg90, Heg04].

In a similar spirit, Johnson and Rosebrugh develop a categorical approach to the view update problem [JR08]. The authors actually design a general framework for database interoperability and support of view updates, based on categorical algebra: the *sketch data model*. They claim successful applications of their approach in consultancies for australian governmental agencies.

Kotidis et al. [KSV06] adopt a radically different approach to the view update problem in the relational setting: they separate the data into a physical layer containing the base tables, and a logical layer containing the tables as observed by the user. The deletion of a tuple on the view does not delete the tuple from the source but makes it invisible to the user. Deletions of tuples from the view are therefore handled differently from deletions of tuples from the base tables. The insertion of a tuple in a view, however, will induce the insertion of tuples in the base tables. The model stores several clones of each tuple in the physical layer, distinguished by unique identifiers. As observed by Fegaras [Feg10], this model that stores additional data beyond the source document can be viewed as a trade-off between independent, materialized views and non-materialized views that would test if the updates can be translated without side effects.

The View Update Problem in XML The results discussed so far concerned the view update problem in general or in the relational setting. There have been several approaches dedicated to the view update problem for XML documents. Vercammen presents a survey on the view update problem for XML, drawing parallels with the relational setting [Ver05]. The view update problem appeared early on in the XML context [AAC+99], although this paper focuses on the introduction of active features in XML, not the view update problem. It only authorizes simple view updates that always result in unambiguous translations.

Foster et al. [FGM+07, FPZ09] study so called *lenses*. These are bidirectional tree transformers (view definitions) that provide two operations: get and put. The *get* operation allows to compute an abstract view of a concrete tree. The *put* operation takes an updated version of the abstract view, together with the original concrete tree, and correspondingly updates the original tree. This way the view definition itself allows to compute the update propagation. In contrast with our approach, views are always materialized. The expressiveness of lenses and of the views defined in our framework (obtained by selecting nodes through XPath queries) are incomparable. Lenses allow e.g. reordering of siblings, which is not possible for our approach. On the other hand, the visibility of a node in our approach is defined by any regular condition on the tree, whereas it only depends on a bounded neighborhood for lenses. Lenses form a general framework, however, and several kinds of lenses have been defined, most of them not specific to trees. For instance, Barbosa et al. [BCF+10] extend the basic lenses with mechanisms for specifying alignments between the document and its view. Those matching lenses can be instantiated with arbitrary alignment functions whose design depends on the applicative context. Matching lenses also integrate a notion of complement, which is used together with the updated version of the abstract view by the *put* function in place of the concrete document. An additional *res* operation allows to compute the value of the complement from the concrete document. In the lenses we discussed so far, one side of the lens, the view, is assumed to be “smaller” than the other. Hofmann et al. generalize the framework to symmetrical lenses, using the notion of complement also [HPW11]. In view of the categorical interpretations of the constant complement and view update approaches surveyed above, it may be interesting to observe that the whole theory of lenses also admits a natural interpretation in terms of categorical operations [HPW11].

The paradigm of lenses, namely bidirectional programming, has also been applied directly to the XQuery language. Liu et al. [LHT07] propose to propagate view updates by defining a *backward* semantics of XQuery expressions. Essentially, the backward semantics of an XQuery expression used to define a view is a function which takes the original source document with the modified view and returns an updated source document. The class of views is incomparable with ours, as for instance it allows copying. Because of copying, update propagation is not necessarily side-effect free. Moreover, as for lenses, it requires materialization.

Several authors consider the problem of updating XML views of relational databases [WRM06, BDH06, CCFV08, Feg10]. For instance, Braganholo et al. [BDH06] focus on translating XML view updates to relational view updates and delegating the problem to the relational DBMS, whereas Wang et al. [WRM06] study the conditions under which a view update is translatable, and extend their result in a subsequent paper [WJRM08]. They tackle the question of *uniform updatability* with a two phase approach: the translation algorithm first exploits the schema and current state of the view to decide if the view update is uniformly updatable, i.e., is translatable from every possible source document. If not, the algorithm tests if the update is never translatable. In the remaining case, the algorithm takes the source document into account. Choi et al. [CCFV08] provide algorithms for the translation of a rich class of view updates. There exist numerous approaches storing XML documents in relational databases, e.g. [TBS02, BGvK$^+$06], and one could attempt to combine them with the view propagation solutions.

It has been argued that XML schemata are more complex than their SQL counterparts, due to richer cardinality constraints and recursive typings [JWMR07], and also due to the hierarchical structure of the document and restructuring capabilities of XQuery [WJRM08]. Jiang et al. [JWMR07] propose solutions to handle those features in the view update problem for XML. They provide an algorithm for the translation of a delete operation over a view defined in a fragment of XQuery. Insertions are out of the scope of the paper. Their schema-based algorithm relies in part on an algorithm proposed by Keller in the relational setting [Kel85].

Reasoning about the Evolution of a Document Several approaches have been proposed to study properties of evolving documents. Cautis et al. [CAM09] introduce update constraints defined by XPath queries. Each XPath constraint comes with an update type: *no-insert*, *no-remove*, or *immutable*, meaning that the set of nodes selected by the query should respectively shrink, grow, or remain the same after application of the updates. Nodes are distinguished by their unique identifier, in a way quite similar to the presentation in this dissertation. Cautis et al. study for various combinations of fragments and update types the problem of deciding if a given constraint is implied by another set of constraints. For fragments of XPath($\Downarrow^*, \Downarrow, [\,], \wedge$), the complexity for the implication problem ranges from PTime to NExptime depending on the fragment and the update types allowed.

Bojańczyk and Figueira define a temporal logic to describe properties satisfied by the evolution of a document tree [BF11]. Essentially, this logic combines a temporal logic that navigates between the nodes of the document using descendant and sibling orderings, with a temporal logic that travels in the time dimension. During the evolution of the document, the domain, i.e., the set of nodes in the trees, is kept constant and only the labeling of nodes varies. An empty document, for instance, is a document of which every node has a "blank" label. The authors show it is easy to evaluate a temporal logic formula of their language over a sequence of documents $t_1, \ldots, t_k$ in time $O(k \times n)$, where n is the size of the domain. More interesting is the *incremental evaluation problem*: the document is initially empty, then nodes are inserted, deleted or relabeled [5]. Each update operation consists in relabeling one node, and is therefore described by a pair $(label, node)$. This guarantees, of course, the size of the domain to be smaller than k. The *incremental evaluation problem* takes as input a formula together with a sequence of k editing operations, and decides if the corresponding sequence of trees – beginning from the empty document – satisfies the formula. The main contribution of the paper is an algorithm solving the incremental evaluation problem with data complexity $O(k \log(n))$, or equivalently $O(k \log(k))$. The query complexity, however, is non-elementary. In its current version, the algorithm only works for trees of bounded depth as it involves a reduction to the word case, but the authors hope to adapt the algorithm to arbitrary trees using forest algebras.

Annotations are another way to trace the history of data. Provenance techniques are quite popular, and find applications in privacy [DKM+11] and more generally security [Che11]. Provenance techniques do not belong to the scope of this paper, but we can still refer the reader to [CCF+09] for an entertaining vision of provenance and its role. Let us also mention two papers that highlight different connections between provenance and update problems: Cong et al. [CFG+11] study the maintenance of annotations under view updates for select-project-join-union views of relational data, focusing on side effect problems, whereas Fegaras [Feg10] exploits provenance (lineage) information to tackle the view update problem for XQuery views and updates of relational data.

Specific Questions Raised by Access Control on Write Operations: Proving Properties of Policies Fundulaki and Maneth [FM07] study the problem of consistency for access control policies on update operations. They define a policy as a set of positive and negative rules allowing or forbidding some update operations. The rules are essentially given by triples consisting of (1) an XPath expression to represent the nodes affected by the rule, (2) an action, such as `insertBefore`, `replace` or `delete`, and (3) an effect which can be + or −, that authorizes or forbids the action to take place at the nodes selected by the XPath expression. A policy is *consistent* if there is no sequence of authorized update operations that is equivalent to a forbidden

[5] actually, the algorithm in the paper focuses on relabelings, but the authors explain that deletions and insertions can be supported using the node identifiers

update. The authors prove that consistency quickly becomes undecidable in presence of both negative and positive rules when the depth of the document is not bounded. For access control rules defined by non-recursive annotated DTD, namely an adaptation of the annotated DTDs of Fan et al. [FCG04] for write operations, Bravo et al. [BCF07] show that consistency can be decided in polynomial time. They furthermore provide an algorithm for repairing inconsistent DTDs. Those results lead to an implementation [BCF08].

From a radically different perspective, Dougherty et al. [DKKdO07] formalize XACML policies by *term rewriting* rules. This formalization allows to apply standard rewriting techniques to reason about properties of policies such as consistency, or for studying the effect of combining several policies. Dougherty et al. [DFK06] also investigate comparison of policies, in a dynamic setting. Their comparison is also based on containment, but the essential effort in the paper consists in taking the environment into account: they argue that the environment is crucial to support credentials, provisional authorization, etc.

Jacquemard and Rusinowitch [JR10] model update policies as term rewriting systems, and focus on forward and backward typechecking algorithms. An update policy is modeled as a term rewriting system parameterized with a standard hedge automaton: the term on the right hand side of a rule may have leaves labeled by states of the automaton, the semantics of such a rule being that a leaf labeled with a state q is replaced by any ground tree accepted by the automaton from state q. They consider a model of hedge automata where the transitions may use context-free languages instead of regular languages to describe the states assigned to the children of a node. The authors define a first class of update rules with this formalism, corresponding to the update primitives of the XQUF, except that the nodes on which the update may be applied can only be specified by their label, whereas XQUF allows to use XQuery to specify on which nodes an update applies. The authors define a second, more expressive class of rules that among others extends XQUF primitive operations with the possibility to delete internal nodes, using the classical adoption mechanism for their children. The forward typechecking problem asks whether any document obtained from some input regular language by (iterated) application of the rewriting rules belongs to some output regular language. This problem is EXPTIME-complete for both classes of update rules, and even PTIME-complete when the output regular language is given by a complete deterministic automaton. This is because the set of documents obtained from a regular language by application of a set of rules of the first class (resp. of the second class) is accepted by a hedge automaton (resp. a context-free hedge automaton), computable in polynomial time. The authors study similarly backward inference, and also investigate an extension of parameterized rewriting systems that allows to specify the nodes on which an update may be applied. An access control policy can be modeled by a pair of rewriting systems, one specifying authorized updates and the other one specifying the forbidden operations. The authors study the complexity of deciding consistency for the two classes of rules under several settings.

Efficient Evaluation of Updates and Transductions We do not investigate in this dissertation the modus operandi for evaluating updates on a document. Some authors however have developed techniques for the efficient evaluation of XQUF updates. For instance, Boncz et al. adapted optimization techniques from XQuery engines [BFG$^+$06], whereas Cavalieri et al. [CGM11] investigate efficient manipulations of pending update lists.

We are not aware of many works for the efficient evaluation of functional non-deterministic transducers, even in the word case, except for a new algorithm by Mohri to disambiguate finite automata and functional transducers [Moh12]. The efforts seems to have focused so far on normalization and determinization [BC02, AM04a]. Mohri and others have investigated weighted (word) transducers for applications in speech recognition. Allauzen and Mohri [AM04a] propose an algorithm that transforms weighted word transducers into determinizable weighted word transducers and report substantial speedup on their experiment. Beal and Carton [BC02] study the determinization of functional word transducers. Choffrut [Cho77] gives a characterization of subsequential functions, defined grosso modo as the transductions that can be accepted by transducers that are deterministic with respect to the input. Weber and Klemm prove that subsequentiality of a transduction can be decided in PTime [WK95].

Filiot et al. [FGRS11] investigate which functional transductions can be evaluated by visibly pushdown transducers using limited memory. More precisely, they introduce two classes of transductions: bounded-memory transductions (BM) and height bounded memory (HBM). BM transductions can be evaluated using space bounded by the size of the transducer only (and thus independent of the input). This class corresponds to subsequential transducers over standard (non-nested) words. The more general HBM transductions can be evaluated with a memory that depends only on the nesting depth (height) of the input and the size of transducer. The authors show that in this second case the memory required is at most exponential in the nesting depth of the input. They provide a general space-efficient algorithm for the evaluation of functional visibly pushdown transducers, and give characterizations for BM and HBM that can be decided in co-NP. Finally, the authors provide a sufficient condition over HBM transduction that guarantees evaluation uses memory quadratic in the nesting depth of the input. This last class of transductions contains and generalizes the determinizable visibly pushdown transductions.

Automata for editing XML documents Shoaran and Thomo [ST11] propose a VPA-based framework to support insertions and deletions in an XML file. Their framework shares several common features with our results on views updates. They consider deletion and insertions of whole subtrees. Similarly to our restrictions to that we call “k-interval-bounded” and “k-synchronized” editing scripts, they bound by a constant k the number of operations applied on the tree, which guarantees polynomial algorithms (with complexity exponential in k if k is not fixed). They use classical constructions on automata to show that their operations preserve regularity, namely: the *deletion of* L' *from* L and the *insertion of* L' *into* L are still visibly pushdown languages for any visibly pushdown languages L and L'. The deletion of L' from L, for instance, consists of all the trees obtained by removing from some tree of L one subtree belonging to L'. In order to define more expressive transformations, the authors generalize these to automata accepting the deletion (resp. insertion) of up to k subtrees. As these operations cannot express conditions on the context in which the subtrees can be deleted, the authors suggest to use XPath formulae to specify nodes at which each deletions and insertions can be applied. Those expressions are then converted into VPAs. In order to apply in parallel several deletions and insertion operations expressed by visibly pushdown automata, the authors propose a multiple phases approach that preliminary marks the nodes to be deleted using visibly pushdown transducers. The deletions and insertion operations are then processed in a second time.

2.5. Schema Approximation

In this section we first survey some results in the literature that deal with the approximation of XML schema or the approximation of context-free language. We finally mention some statistics about schemata and XML documents from the web.

Approximation of Schemata and of Context-Free Languages The statistics surveyed on page 43 show that many XML documents on the web do not refer to any particular schema. This may be one of the reasons why inference of schema has early on been an active topic of research. Recently, Bex et al. [BGNV10, BNSV10] proposed new algorithms for the inference of schemata and investigate their performance experimentally. Due to the locality of DTDs, DTD inference immediately reduces to the inference of deterministic regular expressions. Inference of XML schemata is slightly more intricate due to the subtyping mechanism, but it also reduces to the inference of one deterministic regular expression for each context in which the element appears, the context being the path from the element to the root of the tree [BNV07].

Bex et al. [BNSV10] provide several algorithms for the inference of single occurrence regular expressions (SOREs) and chain regular expressions

(CHAREs). SOREs are the (necessarily deterministic) regular expressions in which each letter occurs at most once in the expression, whereas CHAREs form a subclass of SOREs with very simple structure. Statistics gathered by the authors show that CHAREs represent a huge majority of expressions occurring in practice. Bex et al. extend in [BGNV10] the fragment to k occurrence regular expressions. They show that contrary to deterministic regular expressions, k occurrence regular expressions are learnable in the limit from positive examples, and provide a corresponding algorithm iDRegEx for the inference of deterministic regular expressions. We refer the reader to the above papers for other references on DTD and schema inference.

In this dissertation, we consider the case when a schema is already known but the schema is too complex or is not a DTD nor even an XML Schema. One possible motivation for approximating a DTD or XML Schema lies in the difficulty to handle the determinism requirement from these schema languages, since deterministic regular expressions are less expressive than (standard) regular expressions. Ahonen proposes an algorithm based on the BKW test presented in Section 6.2.2 to approximate an arbitrary regular expression with a deterministic regular expression. Bex et al [BGMN09] propose an optimization of that algorithm together with a new algorithm, and compare experimentally the three algorithms on synthetic regular expressions.

XML Schema is not closed under union and difference operations. Gelade et al. [GIMN10] investigate XML Schema approximations for those operations. They also investigate the approximation of a regular tree language with XML Schema. The XML Schema language being rather complex, however, the authors do not actually consider XML Schema but consider single-type regular EDTDs instead, with content models (productions) given by DFAs. The authors prove that for any EDTD, one can compute a minimal upper-approximation by single-type EDTD in exponential time, and the associated decision problem, i.e., deciding if a given single-type EDTD is the minimal upper-approximation of another, is Pspace-complete. Similarly, one can compute a single-type EDTD for the minimal upper-approximation of the union and intersection of two single-type EDTDs in quadratic time. An upper-approximation for the difference of two single-type EDTDs can similarly be computed in polynomial time. The authors also show there is no unique maximal under-approximation in general.

Whereas Gelade et al. consider union and intersection operations, we essentially consider schemata obtained after deletion of internal nodes, which results in tree languages that need not even be regular in general. We are therefore interested in the approximation of context-free languages with regular languages. Approximations of context-free languages have been motivated by applications in natural language processing and especially for speech recognition, as well as in verification. In the context of natural language processing, Nederhof [Ned00] surveys regular over- and under-approximations of context-free grammars and proposes some new ones. He also evaluates empirically the effectiveness of those approximations regarding among others the size of the automaton obtained and the percentage of sentences from the corpus that are correctly recognized. Some of the approximation techniques surveyed simplify the grammar rules in order to prevent "self-embedding", others restrict the stack behaviour of the corresponding pushdown automaton, and others are based on N-grams, i.e., the factors of size N in the words accepted by the grammar... Mohri and Nederhof [MN00] propose another construction based on the decomposition of the grammar graph into strongly connected components, and which builds in linear time a compact representation of the resulting automaton.

In the context of verification, Ganty et al. [GMM10] investigate under-approximations of context-free languages with bounded languages, i.e., context-free languages that are a subset of $w_1^* w_2^* \dots w_k^*$ for some natural k and some words $w_1, \dots, w_k$. The authors show that each context-free language L admits a subset $L' \subseteq L$ that has the same Parikh image as L and that is a bounded language. They apply their under-approximation technique to test emptiness of the intersection of context-free languages and to compute the reachable states of a program. With Farré and Galvez, Schmitz [GSF06, Sch07] develops a general framework for approximating context-free grammars. This framework is based on the *position graph*, a representation of the set of all derivation trees of the grammar. The *position automaton* is the NFA obtained by quotienting the vertices of position graph with an equivalence relation of finite index. The position automaton therefore depends on the choice of equivalence relation: the coarser the relation, the coarser the approximation. This general framework provides techniques to detect ambiguity of grammars [Sch07] and to cope with non-determinism in parser generation [GSF06].

Statistics on Real Documents and Schemata What are the main characteristics of XML documents on the web? Numerous studies have gathered statistics from the XML web, but we only survey a few of them. Grijzenhout and Marx [GM11] study the quality of documents on the web with respect to validity. They gather a collection of 180 000 XML documents from about 100 000 websites, for a total of 40GB. Of those documents, more than 85% are well-formed, but the proportion is much lower when considering only the documents referencing a DTD. Roughly a quarter of the collection files references a downloadable DTD or XML Schema, but only a third of those (8.9% of the total collection files) validates with respect to their schema. For those referencing a DTD, the failure to validate is explained in 73.5% of the cases by the document's not being well formed, and in 22.3% by a failure to validate the DTD, the remaining percents corresponding to syntactically incorrect DTDs. For those referencing an XML Schema, the failure to validate is explained in 66.5% of the cases by a failure to validate the DTD, and in 31.2% by syntactically incorrect schemata, the remaining percents corresponding to documents that are not well-formed.

Martens et al. [MNSB06] compare DTDs and XML Schemata from both theoretical and practical point of view, using a collection of 819 XML Schemata. In a nutshell, they observe that the additional expressiveness of XML Schemata over DTDs is used to a "very limited extent": out of the 225 correct schemata in the collection, only 15% cannot be expressed with an equivalent DTD. In an overwhelming majority of these non-local languages, the type depends only on the parent context. Bex et al. [BNdB04] makes some similar observations, focusing on the structure of the regular expressions in the content models. They observe that regular expressions occurring in real-life DTDs and XML Schemata are actually very simple.

Barbosa et al. [BMV06] analyse a collection of about 190 000 documents from about 19 000 web sites, for a total of 843MB. They study a whole range of properties including statistics about text nodes and attributes, the size and depth of the documents, fan-out (i.e., number of children) of the element nodes. This study reveals that 99% of the documents in the collection have depth at most 8, with an average depth of 4, and a maximal depth of 135. Three quarters of the documents contain elements with mixed content, i.e., have both text and element-nodes descendants. Those mixed contents account for 5% of all nodes. Another conclusion from the study is the predominance of structural (markup) content over textual content, except for big documents: the content/markup ratio increases with the size of the document. We define as recursive a label common to two elements of which one is an ancestor of the other. The authors observe that about 15% of the documents contain a "recursive" label. Those documents generally do not refer to any DTD. Of those documents, 98% contain a single recursive label, and another 1% contain two recursive labels, with a maximum reached with 9 recursive labels.

Choi [Cho02] uses a sample of 60 DTDs, and observes structural properties of theses, regarding content model, recursivity, etc. Half the DTDs (35) from this small sample are recursive. Most non-recursive DTDs have depth at most 8, with a maximum of 20. Choi also counts the number of simple cycles made possible by the recursive DTDs, where a simple cycle is a sequence of distinct element names apparently representing the possible element names on the path from the root to a node. Half (19) of the recursive DTDs admit at most 10 different simple cycles, 8 recursive DTDs admit between 10 and 100 different simple cycles, and the remaining 8 DTDs admit more than 100 different simple cycles, for a maximum of about 1500.

3. Models for XML Reasoning

You can only find truth with logic if you have already found truth without it.

(G. K. Chesterton, The Man Who Was Thursday)

Contents

3.1. Words, XML, and Unranked Trees

XML, a Text Format that Emphasizes the Structure of the Document The Extensible Markup Language (XML) is a text format defined by W3C specification [XML99]. The origins of this language can be traced back to the SGML language, XML was developed for the purpose of large-scale electronic publishing. The main design goals for XML include usability over the

Internet, compatibility with SGML and facilitating automatic processing of the documents [XML99].

Each XML document contains elements, delimited by start-tags and end-tags. Unlike other markup languages like HTML, XML requires that documents are well-formed: each start-tag has an explicit corresponding end-tag. Furthermore, those tags must be properly nested, with no overlapping. Therefore, an XML document can be represented as a tree, and the construction of this tree is straightforward. Each element n is represented as a node, and the elements between the start- and end-tag of n constitute the subtree below n. In addition to the nesting of elements, XML allows the markup tags to contain attributes, and character data may be inserted between the markup tags. Those could be embodied by leaf nodes in the tree representation. XML belongs to the family of semistructured data models, and XML data is often referred to as redundant and self-describing [XML][1], which comes at the expense of concision. This waste of space is arguably balanced by improved interoperability, a requirement in web applications, and also mitigated by the ever-decreasing cost of storage.

The author of an XML document can define his own element names, therefore the number of XML elements is potentially unbounded. Thus, an essential feature of XML is the use of schemata and namespaces. Namespaces allow to use element names from different vocabularies, avoiding name clashes [XML99], while schemata express constraints on which elements should appear in the document, and where. In this dissertation we do not consider namespaces. But schemata are at the core of most of our algorithms, and we will represent them using logical or automata-theoretic formalisms.

The W3C developed several query languages to extract information from XML documents. The XML Path language (XPath) has proved a fundamental tool for extracting nodes from the document, and it is at the core of more expressive query languages such as XQuery and XSLT. It is common practice in the community to abstract from the arithmetic operations in XPath and consider only the navigational core of XPath, called NavXPath in [BK08] and the present dissertation, and also known as CoreXPath 1.0, although the original definition of CoreXPath does not contain the next- and preceding-sibling axes [GK02]. CoreXPath queries can be easily translated into first order logic formulae, and therefore standard decision problems can be solved using formal methods, whereas arithmetic operations would make most problems undecidable. We follow this classical approach and model queries with tree automata.

[1] Although the description of XML as a self describing language drew much criticism on the web

3.1.1. General Notations and Tools

In this dissertation, $|S|$ will denote the size of S for every object S, and the cardinality of S if S is a set. Also, given a set S and function f, $f(S)$ will be used to denote the set $\{f(x) \mid x \in S\}$ whenever this meaning is clear from context. For a one-to-one function f, the inverse of f will be denoted by f^{-1}: for all x, y, $y = f^{-1}(x)$ iff $f(x) = y$. Similarly, given a binary relation R, we denote by R^{-1} the inverse relation, namely $(x, y) \in R^{-1}$ if and only if $(y, x) \in R$. We define a *binary relation* between two sets S_1 and S_2 as a subset of $S_1 \times S_2$. A binary relation over S_1 is a subset of $S_1 \times S_1$.

Operations on Binary Relations Let us begin with some preliminary remarks on the composition of binary relations. We do not specifically assume binary relations to be represented as adjacency matrices: they are generally lists of pairs, but a matrix representation for $R_1 \subseteq S_1 \times S_2$ can be computed in time $|S_1| \times |S_2|$. The composition of two relations $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ can be obtained by first computing the "join" of R_1 and R_2, and then projecting the join attribute, i.e., the component in S_2. Computing joins is harder than computing composition insofar as the join of two relations may have cubic size. Joins of relations can be computed using sorting and hashing techniques, among others. One can also use an array of size S_2 to compute the join. The composition of $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ can therefore be computed in time $O(|R_1| \times |R_2| + |S_2|)$.

The composition of binary relations can also be interpreted as the multiplication of boolean matrices. Chandra and Merlin advocated as early as 1977 the use of (sparse) boolean matrix multiplication to optimize joins of relations [CM77, p. 80]. In particular, the composition of binary relations over some set S can be viewed as the product of square Boolean matrices of dimension $n = |S|$, and therefore has complexity $O(n^\omega)$ for some $\omega < 2,38$ [CW90].

We henceforth denote by ω this constant that gives the degree of Boolean matrix multiplication[2]. Recent results have slightly improved the value of ω for the Coppersmith-Winograd algorithm, yet this algorithm is notoriously inefficient in practice. Other subcubic algorithms such as Strassen's [Str69] in $O(n^{2,807})$, however, outperform the naïve cubic algorithm for the multiplication of (reasonably) large-dimensional matrices. The comparative performances of matrix multiplication algorithms are controversial, though, as they may be blurred by optimization techniques that drastically improve cache performance or parallelization.

Transitive closure and composition have essentially the same complexity [Mun71, FM71], so that one can also compute in $O(n^\omega)$ the transitive closure of a binary relation over a domain of cardinality n.

[2] the best value for ω to the best of our knowledge is slightly below $2,3727$ [Wil12]

Lemma 3.1. *The composition of two binary relations over a set of cardinality n can be computed in $O(n^{\omega})$. The transitive closure of a binary relation can be computed with the same complexity.*

More generally, let Q and Q' two sets with respectively n and m elements. Given two relations $R_1 \subseteq Q \times Q'$ and $R_2 \subseteq Q' \times Q$, one can compute the composition of R_1 and R_2 in time $O(\lceil m/n \rceil \times n^{\omega})$. This complexity is slightly better than $O(m \times n^2)$ and can be obtained through the decomposition of matrices into blocks of size $n \times n$. Smarter algorithms have been devised to adapt the fast matrix multiplication technique to rectangular matrices [HP98].

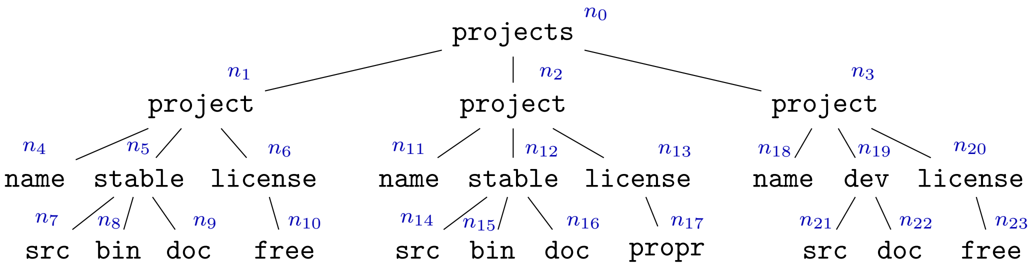
Some algorithms take advantage of particular properties of the matrices to compute more efficiently the product. This is typically the case of output sensitive algorithms [AP09, Lin11], or algorithms for the multiplication of sparse matrices [YZ05]. Nevertheless, as the benefits from fast multiplication algorithms are not very impressive in general, we will use the coarser cubic estimation for the evaluation of algorithms in this dissertation, but will typically mention the potential improvements that could be gained from those smarter algorithms.

Lemma 3.2. *Let S_1, S_2, S_3 three sets. The composition of $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ can therefore be computed in time $O(|R_1| \times |R_2| + |S_2|)$. It can also be computed in time $O(|S_1| \times |S_2| \times |S_3|)$. When $S_2 = S_1$, the transitive closure of R_1 can be computed in $O(|S_1|^3)$.*

3.1.2. Words and Trees as a Model for XML Documents

An *alphabet* Σ is a finite set of letters. The size of alphabet Σ is its number of elements, denoted $|\Sigma|$. A *word* w over alphabet Σ is a sequence $w = a_1 a_2 \ldots a_n$. The *size* of w is $|w| = n$, and we denote by $|w|_a$ the number of occurrences of letter a in w. We also denote by $w[i]$ the letter a_i at position i in w, and denote by $m[i..j]$ the subword $m[i]m[i+1]\ldots m[j]$ of m.

Trees In this dissertation, we model XML documents as unranked ordered trees over a finite alphabet Σ. A tree t is a relational structure $(\Sigma_t, N_t, \mathit{child}_t, \mathit{follow}_t, \mathit{lab}_t)$ where Σ_t is the alphabet, N_t is the set of nodes, $\mathit{lab}_t : N_t \to \Sigma_t$ is the labeling function, $\mathit{child}_t \subset N_t \times N_t$ is the child relation, and $\mathit{follow}_t \subset N_t \times N_t$ is the following sibling relation. Thus $(n, n') \in \mathit{child}_t$ if and only if n' is a child of n in t. We write $n \preccurlyeq_t n'$ if n is an ancestor of n' in t: $\preccurlyeq_t$ is the transitive reflexive closure of child_t. The root of t will be denoted by root_t, while T_Σ denotes the set of trees over the alphabet Σ. The *size* of the tree is its number of nodes: $|t| = |N_t|$. The *leaves* of the tree are the nodes without children. The other nodes are *internal nodes*. Given $n \in N_t$, the *subtree* below n in t is denoted by $t_{\restriction n}$ and is defined as $t_{\restriction n} = t'$ with $\Sigma_{t'} = \Sigma_t$, $N_{t'} = \{n' \in N_t \mid n \preccurlyeq_t n'\}$, $\mathit{child}_{t'} = \mathit{child}_t \cap N_{t'}^2$, $\mathit{follow}_t = \mathit{follow}_t \cap N_{t'}$, and, for every $n' \in N_{t'}$, $\mathit{lab}_{t'}(n') = \mathit{lab}_t(n')$.



$N_{t_0} = \{n_0, n_1, \ldots, n_{23}\}$, $\quad root_{t_0} = n_0$, $\quad lab_{t_0} = \{(n_0, \texttt{projects}), \ldots, (n_{23}, \texttt{free})\}$,
$child_{t_0} = \{(n_0, n_1), (n_0, n_2), (n_0, n_3), (n_1, n_4), (n_1, n_5), (n_1, n_6), \ldots (n20, n23)\}$,
$follow_{t_0} = \{(n_1, n_2), (n_1, n_3), (n_2, n_3), (n_4, n_5), (n_4, n_6), (n_5, n_6) \ldots (n21, n22)\}$.

Figure 3.1.: A tree t_0

We also define other axes to simplify the navigation inside the tree: $next_t$ will denote the next-sibling axis: $(x, y) \in next_t$ if $(x, y) \in follow_t$ and there is no z such that both $(x, z) \in follow_t$ and $(z, y) \in follow_t$. $Parent_t(n)$ represents the parent of node n: $x = Parent_t(y)$ if $(x, y) \in child_t$. Let t a tree, $n \neq n'$ two nodes of t, and let n_0 denote the lowest ancestor of n and n'. Node n comes before (or is smaller than) n' in *document order* if and only if its opening tag comes before the opening tag of n' in the linearization of t. Equivalently, node n is smaller than n' iff one of the two following conditions is satisfied: (1) $n \preccurlyeq n'$ or (2) there exist two nodes n_1, n'_1 such that all following conditions are satisfied: $(n_0, n_1) \in child_{t'}$, $(n_0, n'_1) \in child_{t'}$, $n_1 \preccurlyeq_t n$, $n'_1 \preccurlyeq_t n'$ and $(n_1, n'_1) \in follow_t$.

Example 3.1. *Figure 3.1 contains an example of a tree representing an XML database with information on software development projects. Every project has a name and a type of license (either free or proprietary). Projects under development come with their source codes and documentation, whereas stable projects have also binaries. In tree t_0, the descendant and next-sibling relations are respectively $\preccurlyeq_{t_0} = \{(n_0, n_1), \ldots, (n_0, n_{23}), (n_1, n_4) \ldots\}$ and $next_{t_0} = \{(n_1, n_2), (n_2, n_3), (n_4, n_5), \ldots\}$. In t_0, the node identifiers n_i give the document order of the nodes, but in general the node identifiers may be arbitrary.*

Although most of our trees will be unranked trees, we sometimes use ranked trees, especially binary trees: a tree t has rank k if every node of t has at most k children. A binary tree is a tree of rank 2. A *full* binary tree is a binary tree in which every internal node has exactly two children.

The *depth* of a node $n \in N_t$ is the length of the shortest path from n to $root_t$: $depth_t(root_t) = 0$ and, if $(n', n) \in child_t$ then $depth_t(n) = 1 + depth_t(n')$. The *depth* of tree t is the depth of its deepest node: $depth(t) = \max\{depth_t(n) \mid n \in N_t\}$. The *yield* of a tree t is the word formed by (the labels of) the leaves of t, taken in document order. In particular, we will often identify trees of depth one with their yield, as this is a convenient way to represent word languages in our proofs. A tree (resp. word) language over alphabet Σ is a possibly infinite set of trees. A *hedge* is a sequence of trees.

We also define a notion of isomorphism between trees: an *isomorphism* from t to t' is a bijective mapping ϕ from N_t to $N_{t'}$ such that $(\phi(x), \phi(y)) \in \mathit{child}_{t'}$ iff $(x, y) \in \mathit{child}_t$, $(\phi(x), \phi(y)) \in \mathit{follow}_{t'}$ iff $(x, y) \in \mathit{follow}_t$, and $\mathit{lab}_{t'}(\phi(x)) = \mathit{lab}_t(x)$. Two trees t, t' are isomorphic if they can be related by an isomorphism, or in other words, if t and t' are equal when considered as terms. In that case we write $t \simeq t'$. We point out that equality of trees should not be confused with isomorphism: in general $t \simeq t'$ does not imply $t = t'$. For convenience, however, we are sometimes going to present trees using terms. For instance, the tree t in Figure 3.2 corresponds to the term $r(a, f(b, c), g(d))$.

Several alternative representations for unranked trees have been investigated in the litterature. We will make ample use of two of them: the linearization which represents trees as (nested) words, and the first-child-next-sibling encoding which represents unranked trees as binary trees.

Linearization: Nested Words For every alphabet Σ, let $\hat{\Sigma} = \{op, cl\} \times \Sigma$ be the corresponding tag alphabet, where for any label $a \in \Sigma$, (op, a) stands for the *opening* XML tag `<a>` and (cl, a) for the *closing* XML tag `</a>`. The linearization of a tree is defined as follows: $\mathit{lin}(\varepsilon) = \varepsilon$ and $\mathit{lin}\,(a(t_1, t_2, \ldots, t_n)) = (op, a)\, \mathit{lin}(t_1)\mathit{lin}(t_2) \ldots \mathit{lin}(t_n)\, (cl, a)$. Thus, the linearization of a tree t induces a one-to-one mapping from each node of t into a pair of opening and closing tags in $\mathit{lin}(t)$. For instance, the linearization of tree t in Figure 3.2 is $(op, r)\ (op, a)\ (cl, a)\ (op, f)\ (op, b)\ (cl, b)\ (op, c)\ (cl, c)\ (cl, f)\ (op, g)\ (op, d)\ (cl, d)\ (cl, g)\ (cl, r)$.

Binary Encoding for Unranked Trees The Rabin first-child next-sibling encoding *fcns* [Rab68, Koc03] basically encodes an unranked tree over Σ into a binary tree over the alphabet $\Sigma_\perp = \Sigma \uplus \{\perp\}$. All symbols from Σ have in $\Sigma_\perp$ arity 2, and $\perp$ is the sole constant symbol. The *fcns* encoding of a hedge is defined as: $\mathit{fcns}(\varepsilon) = \perp$ and $\mathit{fcns}(a(t_1, \ldots, t_n), t'_2, \ldots t'_n) = a(\mathit{fcns}(t_1, \ldots, t_n), \mathit{fcns}(t'_2, \ldots, t'_n))$. A binary tree over alphabet $\Sigma_\perp$ is *a fcns tree* if it is the *fcns* encoding of some unranked tree, i.e., if it is a full binary tree with internal nodes labeled in Σ, leaves labeled with $\perp$, and such that the right child of the root is a leaf. For instance, the *fcns* encoding of tree $f(a, b, c)$ is given by $\mathit{fcns}(f(a, b, c)) = f(a(\perp, b(\perp, c(\perp, \perp))), \perp)$, and the *fcns* encoding of $r(a, f(b, c), g(d))$ is given on Figure 3.2.

An alternative binary encoding of unranked tree is the curry encoding of terms (also called "extension encoding"), which we will not use. The curry encoding is defined inductively by $\mathit{curry}(a) = a$ and $\mathit{curry}(a(t_1, \ldots, t_n)) = @(\mathit{curry}(a(t_1, \ldots, t_{n-1})), \mathit{curry}(t_n))$. The tree $f(a, b(d), c)$, for instance, is encoded into $@(@(@(f, a), @(b, d)), c)$.

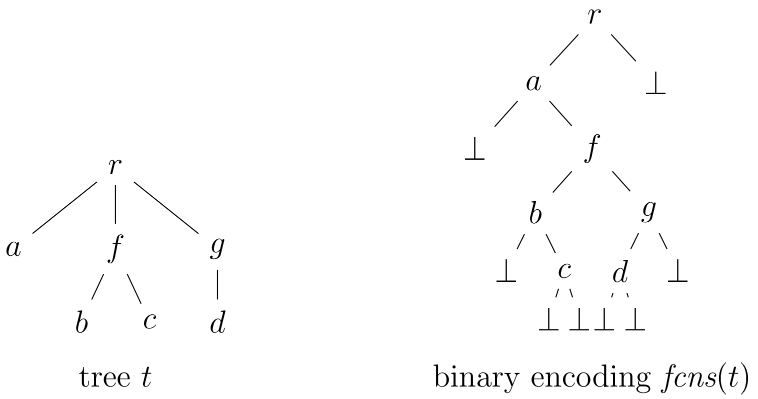


Figure 3.2.: The *fcns* encoding.

Morphisms A *morphism* Φ from alphabet Σ to alphabet $\Delta \cup \{\varepsilon\}$ is specified by a function from Σ to $\Delta \cup \{\varepsilon\}$. The notion of morphism extends this relabeling function to words and trees as follows: the (word) morphism induced by Φ is the function that maps a word $w = a_1 \dots a_n$ to the word $\Phi(w) = \Phi(a_1) \dots \Phi(a_n)$: ε is interpreted as the neutral element of the free monoid and therefore ε symbols are removed in $\Phi(w)$. Similarly, given a tree $t = (\Sigma, N_t, child_t, follow_t, lab_t)$, the (tree) *morphism* induced by f is the function that maps every tree t over Σ into the tree t' over Σ'_ε s.t. t' is obtained from t by relabeling every node $n \in N_t$ as $f(lab_t(n))$, and then deleting the (resulting) subtrees whose root is labeled by ε. In this dissertation, tree morphisms will always be defined in such a way that for every node mapped to ε, all its descendants are also mapped to ε. Thus the morphism for trees corresponds to the morphisms on the linearization: for every morphism Φ and tree t over Σ $lin(\Phi(t)) = \Phi(lin(t))$, with the convention that $\Phi((\eta, a)) = (\eta, \Phi(a))$ for every $\eta \in \{op, cl\}$ and $a \in \Sigma$. We do not distinguish the function f and the morphism it induces.

This notion of morphism can be viewed as a very restricted adaptation of the notion of (ranked) tree homomorphism and can also be viewed as a special case of morphism of forest algebras as defined in [BW08]. Indeed, the morphisms we have defined correspond to a subclass of linear alphabetic (non necessarily non-erasing) tree homomorphisms on the *fcns* encoding. More precisely, let us we denote by f_e the linear alphabetic tree homomorphism [CDG+07] induced by f on the trees over $\Sigma_\perp$. Formally, $f_e(a(t_1, t_2))$ is equal to $f(a)(f(t_1), f(t_2))$ if $f(a) \neq \varepsilon$, and is equal to ε otherwise. Then $f_e(fcns(t)) = fcns(f_e(t))$. We get easily the following result.

Proposition 3.3. *The image and the inverse image of a regular set of trees under a morphism are regular sets of trees.*

Proof. The proof is easily obtained by using the *fcns* encoding and the closure properties of regular ranked tree languages under inverse morphisms and linear morphisms. The constructions are polynomial. □

Limitations with Respect to Real XML Documents Our tree models abstract several features from the XML data model. First of all, we ignore attribute nodes and text nodes. Then we do not represent the prologue of the document: the only part of the document that we model is the body of the XML document. Last but not least, the lack of attributes prevents us from supporting key mechanisms at the schema level. Neither do we model XML namespaces nor DTD entities. On the one hand we attribute identifiers to the nodes of our trees, but on the other hand the queries we consider cannot refer to those identifiers, which anyway have particular meaning (and we never consider rich identifiers schemes).

3.1.3. Regular Expressions and Word Automata

Throughout this thesis we will use several devices such as automata, regular expression and logical formulae in order to define word and tree languages. We will say that any two such devices are *equivalent* if they accept the same language.

Regular Expressions *Regular expressions* are generated by grammar:

$$e \coloneqq \epsilon \mid a \mid (e) \odot (e) \mid (e) + (e) \mid (e)^* \qquad (\text{with } a \in \Sigma)$$

We impose parentheses in order to obtain an immediate construction of the parse tree of e, but we will omit the parentheses whenever we can. The *parse tree* t_e of e is a binary tree, defined as usual, with node identifiers attributed arbitrarily: the parse tree of an expression is only used in Sections A and A, where the parse tree is given as input of the algorithms, so that node identifiers are already provided. A sample expression and its parse tree will be presented in Figure 6.2 on page 206. Each node of t_e represents some subexpression, and the leaves of t_e with label in Σ are the *positions* of e, denoted by $Pos(e) = \{n \in N_t \mid lab_e(n) \in \Sigma\}$.

The symbols $\odot$, $+$ and * represent concatenation, disjunction, and Kleene star, respectively, but we generally omit the concatenation symbol. The language $L(e)$ of all words accepted by a regular expression e is defined as usual:

$$\begin{array}{ll} L(\epsilon) = \{\varepsilon\} & L(e_1 \cup e_2) = L(e_1) \cup L(e_2) \\ L(a) = \{a\} & L(e_1 \odot e_2) = L(e_1) \odot L(e_2) \\ L(e^*) = \bigcup_{i \geqslant 0} L(e)^i & \text{where } L(e)^i = \underbrace{L(e) \odot \ldots \odot L(e)}_{i} \end{array}$$

We define the size of a regular expression as its number of symbols: $|e_1 \odot e_2| = |e_1 + e_2| = |e_1| + |e_2| + 1$, $|e_1^*| = |e_1| + 1$, and $|a| = |\epsilon| = 1$ ($a \in \Sigma$). We sometimes use a set to define the disjunction of elements in that set: for instance, Σ will be used as a shorthand for $+_{a\in\Sigma} a$. For regular expressions defining DTDs, we will use , and | to denote the concatenation and disjunction instead of $\odot$ and $+$.

In Section 6.2 we will discuss regular expressions with numeric occurrences: those are obtained by extending the syntax of regular expressions with $e^{[n..m]}$ for every $n \in \mathbb{N}, m \in \mathbb{N} \cup \{\infty\}$ with $n \leqslant m$. The semantics of these numeric occurrence indicators is given by $L(e^{[n..m]}) = \bigcup_{n\leqslant i\leqslant m} L(e)^i$. In particular, $L(e^*) = L(e^{[0..\infty]})$. The *regular expressions with squares* are (standard) regular expression extended with $e \coloneqq e^{[2..2]}$. The size of a regular expression with numeric occurrence indicators is defined by $|e^{[n..m]}| = |e| + 1$. Squares and even numeric occurrence indicators do not increase the expressivity of regular expressions, but they drastically improve succinctness: the following example gives a family of regular expressions e_n of size n with numeric occurrences such that every (standard) regular expression equivalent to e_n has size at least 2^n.

Example 3.2. *Set $e_1 = a$, and for every natural n, set $e_n = (e_{n-1})^{[2..2]}$. Expression e_n has size n, but every (standard) regular expression with language a^{2^n} has size $\Omega(2^n)$. Conversely, for every regular expression e with squares, one can build a regular expression e' of size at most $2^{|e|}$ such that $L(e') = L(e)$.*

Word Automata Automata are one of the most extensively studied models in formal methods for computer science. A *nondeterministic finite automaton* (NFA) is a tuple $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$ where Σ is the alphabet, Q is a finite set of states, I is the set of initial states, F is the set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the set of rules. The *size* of automaton $\mathcal{A}$ is defined as $|\mathcal{A}| = |Q| + |\Delta|$.

The semantics of finite state automata is given through the notion of *runs*. Given a word $w = a_1a_2\dots a_n \in \Sigma^*$, a *run* of $\mathcal{A}$ on w is a mapping $\rho : \{0,\dots,n\} \to Q$ such that $\rho(0) \in I$ and $(\rho(i-1), a_i, \rho(i)) \in \Delta$ for every $i \in \{1,\dots,n\}$. The run ρ is *accepting* if $\rho(n) \in F$. Automaton $\mathcal{A}$ *accepts* word w if it has an accepting run on w. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all the words accepted by $\mathcal{A}$.

$\mathcal{A}$ is *deterministic* if $|I| = 1$ and, for every $q \in Q$ and $a \in \Sigma$, there exists at most one q' with $(q, a, q') \in \Delta$. It is well known that every NFA can be determinized in time at most exponential, and there are NFAs for which the exponential blowup in the number of states cannot be avoided.

Several word automata models have been proposed to extend the basic automata. Pushdown automata, for instance, use a stack in addition to the finite set of states, while Mealy machines and Moore machines do not only accept an input word, but also produce an output [Mea55, Moo56]. These models are commonly used for the verification of system properties. We also use a model of pushdown automata in addition to standard word automata in this dissertation. Furthermore, our models for document transformations can be viewed as a model of transducers. We shall discuss in the next sections visibly pushdown automata and our modelization of document transformations by tree alignment languages.

Conversions Between Regular Expressions and Word Automata Early works pertaining to the formal study of regular languages and finite automata can be traced back to the first half of the 20th century, for instance, as a model for neural net behaviour [MP43]. A major result by Kleene establishes the equivalence between regular expressions and automata [Kle56] in terms of expressiveness. This result initiated a recurring interest in the connections between regular expressions and automata. While regular expressions are essentially used to describe regular languages, finite state automata are the foremost tool for reasoning about those languages.

There are two major approaches for converting an automaton into an equivalent regular expression. McNaughton and Yamada [MY60] proposed a method based on decomposition of the paths in the transition graph of the automaton. Brzozowski and McCluskey [BM63] proposed the *state elimination method*, which can be applied either by constructing generalized automata in which the transitions are labeled with regular expressions instead of letters, or by solving a system of algebraic equations using Arden's Lemma. We do not detail those algorithms: they are surveyed in [Sak05] and many standard textbooks on automata theory such as [Sak09]. The above algorithms have exponential complexity. A corresponding exponential lower bound has been provided by Ehrenfeucht and Zeiger [EZ74]. They show that for every $n \geqslant 0$ there is a DFA $\mathcal{A}_n$ with n states and size $O(n^2)$ over an alphabet Σ of size n^2 such that any regular expression equivalent to $\mathcal{A}_n$ has size at least 2^{n-1}. Similarly, they show a supra-exponential $n^{\Omega(\log \log n)}$ lower bound for acyclic DFA. Gruber and Holzer improve in [GH08] the $2^{\Theta(\sqrt{n})}$ lower bound for the conversion of DFAs into regular expressions to $2^{\Omega(n)}$, which matches the upper bound of the above algorithms. More precisely, they show with graph-theoretic techniques that for any alphabet Σ of size at least 2, there is a family of DFAs $\mathcal{A}_n$ over Σ with at most n states such that any regular expression equivalent to $\mathcal{A}_n$ has size at least $2^{\Omega(n)}$. Consequently the conversion of automata into regular expressions has complexity $2^{\Theta(n)}$.

Algorithms building finite automata from regular expressions have found widespread applications in domains such as lexical analysis or pattern matching. Several algorithms have been proposed to convert regular expressions into NFAs. The major ones are discussed in most textbooks on automata theory and we will not survey all existing constructions. Assume a regular expression e over alphabet Σ. An approach popularized by Thompson [Tho68] allows to build in linear time $O(|e|)$ an NFA $\mathcal{A}'$ with ε-transitions from e,

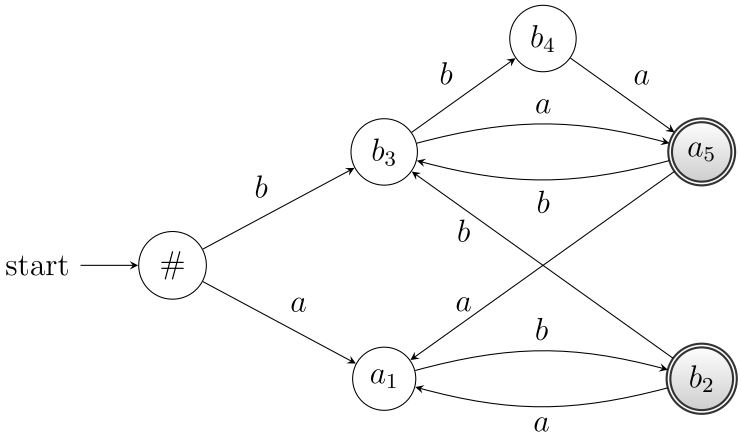


Figure 3.3.: Glushkov automaton of $(ab + b(b + \varepsilon)a)^*$.

but eliminating the ε transitions from $\mathcal{A}'$ raises the complexity to $O(|e|^2)$ if we use the standard algorithm to eliminate the ε-transition [HU79, p. 26]. Another approach is to build (ε-free) NFAs from e. Algorithms following this approach produce quadratic-size (ε-free) NFAs, such as the Glushkov automaton, Follow automaton, or the Antimirov automaton. We detail the Glushkov construction because it enjoys a nice connection to deterministic regular expressions.

Glushkov Automaton The Glushkov automaton of a regular expression has been introduced in [Glu61, MY60]. A striking property of this automaton is a correspondence between the states of the automaton and the occurrences of letters (aka. positions) in the expression. We denote by $\overline{e}$ the regular expression obtained from e by marking the i-th position (from left to right) with subscript i. We denote by $\overline{\Sigma}$ the set of symbols obtained from Σ by adding subscripts below letters. In particular, $Pos(\overline{e}) = Pos(e)$. The First and Last-positions of a regular expression e are $First(e) = \{i \mid \exists u \in \overline{\Sigma}^*.lab_{\overline{e}}(i) \odot u \in L(\overline{e})\}$ and $Last(e) = \{i \mid \exists u \in \overline{\Sigma}^*.u \odot lab_{\overline{e}}(i) \in L(\overline{e})\}$, respectively. Given a position p of e, $Follow_e(p)$ is the set of positions that may follow p in e: $Follow_e(p) = \{q \mid \exists u, v \in \overline{\Sigma}^*.u \odot lab_{\overline{e}}(p) \odot lab_{\overline{e}}(q) \odot v \in L(\overline{e})\}$.

The *Glushkov automaton* of regular expression e is defined as Glushkov$(e) = (\Sigma, Q, I, F, \Delta)$ where $Q = \{\#\} \cup Pos(e)$ (with $\#$ a fresh symbol outside Σ), $I = \{\#\}$, $F = Last(e)$ if $\varepsilon \notin L(e)$, $F = \{\#\} \cup Last(e)$ otherwise, and Δ defined as follows: for every $q, q' \in Pos(e), a \in \Sigma$, (q, a, q') belongs to Δ if and only if $lab_e(q') = a$ and $q' \in Follow_e(q)$. Furthermore, for every $q \in Pos(e)$, $(\#, a, q)$ belongs to Δ if and only if $lab_e(q) = a$ and $q \in First(e)$. Figure 3.3 depicts the Glushkov automaton of regular expression $e_1 = (ab + b(b + \varepsilon)a)^*$.

The Glushkov construction produces quadratic-size automata, albeit with a linear number of states. For instance, over alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the Glushkov automaton for $e = (a_1 + \ldots + a_n)(a_1 + \ldots + a_n)$ has n^2 transitions. Similarly, the Glushkov automaton for $e' = a_1?a_2?\ldots a_n?$ has $n(n-1)/2$ transitions.

It was a long-standing open problem whether an NFA of subquadratic size could be built from all regular expressions, with some people assuming the quadratic increase to be unavoidable, for instance on expression e' (see the discussion in [HSW97, HSW01]). This problem was first solved by Hromkovič et al. [HSW97, HSW01], who presented an algorithm converting any regular expression of size n into an NFA of size at most $O(n(\log n)^2)$. The complexity of the algorithm from [HSW97] is $O(n^2(\log n))$, but Hagenah and Muscholl [HM98] improved the complexity to $O(n(\log n)^2)$. What is more, [HSW97] also contributes a lower bound, as they prove that any NFA equivalent to e' needs $\Omega(n \log n)$ transitions. This lower bound was subsequently improved to $\Omega(n(\log n)^2)$ by Schnitger in [Sch06], still using expression e'. Thus, the $O(n(\log n)^2)$ algorithm from [HM98] is optimal for the conversion of regular expressions into (ε-free) NFAs.

This optimality only applies to the construction of a full ε-free NFA, however: there are other representations of regular expressions that can be computed in linear time, besides the NFA with ε-transitions. Chang and Paige [CP97], on one side, and Ponty, Ziadi and Champarnaud [PZC96] on the other, propose a representation of the Glushkov automaton that can be computed in linear time from the regular expression. This representation allows, given a position n of e, to compute $\mathit{Follow}(n)$ in linear time $O(|\mathit{Follow}(n)|)$. It is interesting to observe that all algorithms in [HM98, HSW97, HSW01, CP97, PZC96] are based on similar observations on the structure of regular expressions.

Determinism of Regular Expressions A regular expression e is *deterministic* if for every position p of e and every $q \neq q' \in \mathit{Follow}_e(p)$ it holds that $\mathit{lab}_e(q) \neq \mathit{lab}_e(q')$ and, for every $p, p' \in \mathit{First}(e)$ with $p \neq p'$, it holds that $\mathit{lab}(p) \neq \mathit{lab}(p')$. In terms of the Glushkov automaton, e is deterministic if and only if its Glushkov automaton is deterministic. Deterministic regular expressions are also called *one-unambiguous* regular expressions in the litterature because of this property which facilitates the evaluation of a word against a regular expression. We say that a regular language L_0 is *deterministic* if there exists a deterministic regular expression e such that $L_0 = L(e)$.

Example 3.3. *Let us define $e_1 = (ab + b(b+\varepsilon)a)^*$ and $e_2 = (a^*ba + bb)^*$. Let us denote by $p_1, p_2, \ldots p_5$ the positions of e_1 in left-to-right order, and by $q_1, \ldots, q_5$ those of e_2. We have $\overline{e_1} = (a_1b_2 + b_3(b_4+\varepsilon)a_5)^*$ and $\mathit{Follow}_{e_1}(p_3) =$*

$\{p_4, p_5\}$. Similarly, $\overline{e_2} = (a_1^* b_2 a_3 + b_4 b_5)^*$, and $Follow_{e_2}(q_3) = \{q_1, q_2, q_4\}$. Regular expression e_2 is non-deterministic since $lab_{e_2}(q_2) = lab_{e_2}(q_4) = b$, but e_1 is deterministic. One can check on Figure 3.3 that the Glushkov automaton of e_1 is deterministic.

We investigate or survey in Chapter 6 a few problems pertaining to deterministic regular expressions, such as deciding efficiently if a regular expression is deterministic, deciding if a regular language is deterministic, and evaluating such regular expressions.

3.1.4. Grammars

Context-free languages and grammars were initially designed as a formal model for natural languages [Cho59]. They were extensively studied in the context of syntactic analysis. In this dissertation we mainly use complexity results on classical decision problems for context-free grammars in order to establish lower bounds.

A *context-free grammar* (CFG) is a tuple $G = (V, T, S, P)$ with V the non-terminals, T the terminals (disjoint from V), S the initial non-terminal, and $P : V \to (V \cup T)^*$ the productions. Given $u, v \in (V \cup T)^*$, we write $u \to v$ if there exist $x \in V$, w_1, w_2 and w' such that $u = w_1 x w_2$, $v = w_1 w' w_2$ and $(x, w') \in P$. The reflexive transitive closure of $\to$ is denoted by $\to^*$. The *language* accepted by a context-free grammar G is a *context-free language* defined as $L(G) = \{w \in T^* \mid S \to^* w\}$. Whenever the context raises no ambiguity, we will speak of context-free languages, assuming implicitly that those languages are given by a context-free grammar. Thus, when we mention the undecidability of universality of context-free languages, we mean: "the problem that takes as input a context-free grammar G over alphabet Σ and decides if $L(G) = \Sigma^*$ is undecidable".

Given a CFG G and a word $w \in L(G)$, a *derivation tree* of w for G is a tree t satisfying the following conditions: the internal nodes of t have label in V, its leaves have label in T, the yield of t is w, and for every internal node of t with label x, if u is the word formed by the children of x, then $x \to u$ belongs to P. A grammar is *ambiguous* if there exists one word $w \in T^*$ that admits two non-isomorphic derivation trees. The problem of testing if a context-free grammar is ambiguous is undecidable.

A *straight line program* is a context-free grammar $G = (V, T, S, P)$ such that there is a single production from each non-terminal, and the production relation is acyclic, i.e., $\forall x \in V, w_1, w_2$, if $x \to^* w_1 x w_2$ then $w_1 = w_2 = \varepsilon$. Thus, each straight line program G represents a single word w_G, of size at most exponential in the size of G.

The Post Correspondence Problem The Post Correspondence Problem (PCP) can be viewed as a problem of deciding ambiguity for particular CFGs.

PCP takes as input an integer n, an alphabet Σ and two sequences of words $u_1, \dots, u_n$ and $v_1, \dots, v_n$. It returns true if there is a PCP match for the two sets of words, i.e., if there are $k \in \mathbb{N}$, $i_1, \dots, i_k \in \mathbb{N}$ such that $u_{i_1} u_{i_2} \dots u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_k}$. Otherwise, it returns false. The Post Correspondence Problem is undecidable.

Properties of Context-Free Languages Context-free languages are strictly more expressive than regular languages. Moreover, they are strongly related to the regular tree languages. Not only because the linearization of a regular tree language is a visibly pushdown language, and visibly pushdown languages are a subclass of context-free languages, but also because the languages formed by the leaves of regular sets of trees are exactly the context-free languages: a set of words S is a context-free language if and only if there exists a regular tree language L such that $S = \mathit{yield}(L)$. Thus, context-free languages appear naturally when we consider regular tree languages and delete internal nodes. However, most decision problems become intractable for context-free languages. What is worse, context-free languages are not closed under all Boolean operations, unlike regular languages or visibly pushdown languages. The intersection of two context-free languages needs not be a context-free language. Nor does the complement of a context-free language need to be a context-free language.

Emptiness can be decided in linear time for a context-free language. But universality is undecidable for context-free languages. Consequently, even the problem of deciding if $R \subseteq L_G$ for R a regular language and L_G a context-free language is already undecidable. Inclusion, and even equivalence of two context-free languages are also undecidable. Testing if the intersection of two context-free grammars is empty is undecidable. Last, it is undecidable if the language of a context-free language is regular. We prove that this undecidability result still holds for deterministic regular languages as the standard proof [HU79] carries over to deterministic regular languages. That determinism of R does not help to test inclusion $R \subseteq G$ is trivial by taking $R = \Sigma^*$.

Proposition 3.4. *The problem of testing if $L(G)$ is a deterministic regular language for a context-free grammar G is undecidable.*

Proof. We simply tailor for deterministic regular expressions the proof of Greibach's theorem [Gre68] in the textbook of Hopcroft and Ullman [HU79, p. 205]. This proof proceeds by reduction from universality of context-free grammars. Let $L_0 = a^n b^n$. Given any context-free language L over alphabet $\Sigma = \{a, b\}$, $L_1 = \Sigma^* \# L \cup L_0 \# \Sigma^*$ is a context free language. Furthermore, L_1 can be effectively computed from L, and L_1 is a deterministic regular language if and only if $L = \Sigma^*$. Note that when $L = \Sigma^*$, L_1 equals $\Sigma^* \# \Sigma^*$

and is therefore deterministic, and otherwise L_1 is not even regular. Consequently, an algorithm testing if L_1 is a deterministic regular language would yield an algorithm to test universality of context-free languages. □

3.2. Tree Languages

3.2.1. Tree Automata and Visibly Pushdown Automata

Tree automata were introduced by Doner [Don65, Don70] and Thatcher and Wright [TW65, TW68] in order to obtain decision procedures for monadic second order logic. Tree automata essentially adapt the word automata models to (ranked) trees: they still use a finite set of states and transition rules, but the transition rules associate the state at a node with the states at its children, instead of associating the state at position i with the state at position $i + 1$.

Tree Automata for Binary Trees Let us define tree automata over (full) binary trees. We only consider automata over *fcns* trees, representing the encoding of unranked trees, so symbols from the alphabet Σ have arity 2 and label internal nodes while leaves are labeled $\perp$. A *(non-deterministic) tree automaton for binary trees (NTA)* over alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ with $\Delta \subseteq Q^3 \times \Sigma \cup Q \times \{\perp\}$. We sometimes write $a(q_1, q_2) \to q$ instead of $(q, q_1, q_2, a) \in \Delta$ and $\perp \to q$ instead of $(q, \perp) \in \Delta$, where $q_1, q_2, q \in Q$, $a \in \Sigma$. The *size* of $\mathcal{A}$ is $|Q| + |\Delta|$. Automaton $\mathcal{A}$ is (bottom-up) *deterministic* if there exists a unique q such that $\perp \to q$ and if additionally for every $q_1, q_2 \in Q$ and $a \in \Sigma$ there exists at most one q such that $a(q_1, q_2) \to q$. Automaton $\mathcal{A}$ is *unambiguous* if for every tree t, $\mathcal{A}$ has at most one accepting run over t.

Given a tree t over alphabet $\Sigma \cup \perp$, a run of $\mathcal{A}$ over t is a mapping ρ from N_t to Q such that for every node n of t, if n is a leaf then $\perp \to \rho(n) \in \Delta$, and otherwise, denoting by n_1 and n_2 the left and right children of n, $lab_t(n)(\rho(n_1), \rho(n_2)) \to \rho(n) \in \Delta$. The run ρ is *accepting* if $\rho(root_t) \in Q_f$. Given any state $q \in Q$, we denote by $\mathcal{A}_q$ the NTA obtained from $\mathcal{A}$ by replacing Q_f with $\{q'\}$. We observe that, when $\mathcal{A}$ accepts only *fcns* encodings of trees, then $\mathcal{A}_q$ accepts *fcns* encoding of hedges.

Of course automata for binary trees can be generalized to automata for ranked trees. The challenges we address, however, are not about ranked but about unranked trees. Several approaches extend the automata framework to unranked trees: a first solution is to encode unranked trees as binary trees, using the fcns encoding for instance. We will occasionally switch to this approach when it helps keep simpler proofs, but will in general adopt a formalism based on the linearization of trees: visibly pushdown automata.

Visibly Pushdown Automata

Visibly pushdown automata (VPAs) have been introduced by Rajeev Alur and Parthasarathy Madhusudan in [AM04b] in order to model program analysis. VPAs are special pushdown (word) automata whose stack behavior is driven by the input symbol according to a partition of the alphabet. Although they were not initially defined for this purpose, VPAs are very useful for processing XML streams, since they can accept well-matched languages defined over an input alphabet of opening tags and closing tags. *Nested word automata* [Alu07, AM09] are a reformulation of visibly pushdown automata. We refer the reader to [AM09, Gau09] for a more detailed analysis of properties of those automata, and their connection to other tree automata representations.

Definition 3.1. *A* visibly pushdown automaton *over alphabet* Σ *is a tuple* $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ *where*

- Σ *is the* input alphabet*,*
- Q *is a finite set of states,*
- Γ *is a finite alphabet of* stack symbols*,*
- $I \subseteq Q$ *is the set of* initial states*,*
- $F \subseteq Q$ *is the set of* final states*,*
- *and* $\Delta \subseteq Q \times \{op, cl\} \times \Sigma \times \Gamma \times Q$ *is the set of* rules.

We shall define hereunder an additional condition that must be satisfied by the VPA $\mathcal{A}$*.*

The *size* of $\mathcal{A}$ is $|Q| + |\Gamma| + |\Delta|$. The states (, letters) and stack symbols that do not occur in transitions of $\mathcal{A}$ can be removed in linear time from $\mathcal{A}$ so we assume throughout the dissertation that the size of a VPA is essentially its number of transition: $|\mathcal{A}| \in \Theta(|\Delta|)$. For the analysis of basic constructions, we will nevertheless distinguish the contribution of $|Q|$, $|\Gamma|$ and $|\Delta|$ in the complexity. This is because $|\Delta|$ and therefore $|\mathcal{A}|$ may contain up to $|Q|^2 \times |\Gamma| \times |\Sigma|$ transitions: the "gap" between state complexity and transition complexity is much wider for VPAs than for word automata.

A rule $(q, \iota, a, \gamma, q') \in \Delta$ is written $q \xrightarrow{(\iota,a):\gamma} q'$. When ι is equal to op, then $q \xrightarrow{(\mathrm{op},a):\gamma} q'$ is a *push rule*. It means that *if the current state is* q *and the input letter is an opening* a *then one can push* γ *into the stack and set the current state to* q'. Symmetrically, $q \xrightarrow{(\mathrm{cl},a):\gamma} q'$ is a *pop rule*. It means that *if the current state is* q *and the top of the stack is* γ *and the input letter is a closing* a *then one can pop* γ *from the stack and set the current state to* q'.

We will sometimes define VPAs with ϵ-transitions of the form (q, ϵ, q') with $q, q' \in \Sigma$ in the rules. This does not increase the expressiveness of the VPAs because the ϵ transitions can be eliminated in polynomial time. To eliminate the ϵ transitions we can add a new rule $(q_0, \iota, a, \gamma, q'_k)$ in Δ for every $(q, \iota, a, \gamma, q') \in \Delta$ and every $j, k \leqslant |Q|$, $q_0, q_1, \ldots, q_j \in Q$ and $q' = q'_0, \ldots q'_k \in Q$ satisfying the following three conditions: (1) $q_j = q$, (2) for every $i < j$, $(q_i, \epsilon, q_{i+1}) \in \Delta$, and (3) for every $i < k$, $(q'_i, \epsilon, q'_{i+1}) \in \Delta$.

Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ be a visibly pushdown automaton, then a *run* of $\mathcal{A}$ from q_0 to q_m over a word $a_1 a_2 \ldots a_m \in (\{op, cl\} \times \Sigma)^*$ is a sequence $(q_0, \sigma_0), (q_1, \sigma_1), \ldots (q_m, \sigma_m)$ with $q_i \in Q$ and $\sigma_i \in \Gamma^*$ for every $i \in \{0, \ldots m\}$, such that $\sigma_0 = \sigma_m = \varepsilon$ and for every $i \in \{1, \ldots, m\}$, there are some $b \in \Sigma$ and $\gamma \in \Gamma$ such that either $a_i = (op, b)$, $(q_{i-1}, op, b, \gamma, q_i) \in \Delta$ and $\sigma_i = \sigma_{i-1} \cdot \gamma$, or otherwise $a_i = (cl, b)$, $(q_{i-1}, cl, b, \gamma, q_i) \in \Delta$ and $\sigma_{i-1} = \sigma_i \cdot \gamma$. The run is *accepting* if $q_0 \in I$ and $q_m \in F$. The pair (q_i, σ_i) is the *configuration* reached by $\mathcal{A}$ on run ρ after reading the i^{th} letter of the word w. By extension, a run of $\mathcal{A}$ over tree t is defined as a run of $\mathcal{A}$ over $lin(t)$. A tree t (or the corresponding nested word $lin(t)$) is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ over $lin(t)$. The language of a VPA $\mathcal{A}$ is a *visibly pushdown language*, and is defined as the set of all trees (or equivalently all linearizations of trees) accepted by $\mathcal{A}$.

Caveat: *In this dissertation, we only consider documents represented as trees, so we require that every word accepted by our VPAs must be the linearization of some tree or hedge: for instance, we consider that a word like $(op, a)\,(cl, b)$ is not accepted by any VPA.* We sometimes assume that the VPAs can accept a hedge instead of a tree, but in this case it will be explicitly mentioned unless it is clear from context. There are two restrictions of the above definitions that guarantee the language only contains trees (or hedges); in this dissertation we assume the second one:

1. We could consider that a VPA $\mathcal{A}$ may have accepting runs over words that are not the linearization of an hedge, although these words do not belong to the language of $\mathcal{A}$: $L(\mathcal{A})$ would be defined as the set of all words w such that $\mathcal{A}$ has an accepting run over w *and* w is the linearization of some hedge.

2. Or we could require that the transitions of the VPA check if the input is the linearization of a hedge. In other words a tuple $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ as above is not a VPA unless it satisfies the property that every word over which $\mathcal{A}$ has an accepting run is the linearization of a hedge.

We call the first and second assumption the *expected tree-input* and *enforced tree-input assumption for VPAs.* We even distinguish the *strong enforced tree-input assumption for VPAs* in which every stack symbol determines the letters which can be processed by a transition: that is, the VPA $\mathcal{A}$ satisfies the strong enforced tree-input assumption if there is a mapping f from Γ to Σ such that $\gamma \in \Gamma$ can only appear in transitions of the form $(q, \iota, f(\gamma), \gamma, q')$ for some q, ι, q'. In this dissertation we use the enforced tree-input assumption, but the results which depend on it will be explicitly mentioned.

Actually, for any VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ over alphabet Σ that does not satisfy the enforced tree-input assumption, we can build a VPA $\mathcal{A}' = (\Sigma, Q, \Gamma', I, F, \Delta')$ in time $O(|\mathcal{A}|)$ satisfying the (strong) enforced tree-input assumption such that the language of $\mathcal{A}$ (under the expected tree-input assumption) is equal to the language of $\mathcal{A}'$. The VPA $\mathcal{A}'$ can be obtained from $\mathcal{A}$ by encoding the letters into the stack symbol: $\Gamma' = \Gamma \times \Sigma$, and $(q, \iota, a, (\gamma, a), q')$ belongs to Δ' if and only if $(q, \iota, a, \gamma, q')$ belongs to Δ. Thus, using the enforced input-tree assumption does not limit the expressive power of our VPA w.r.t. tree (hedge) languages, and does not impact the size of the VPA although it impacts the number of stack symbols. We shall also prove that the problem of checking if a VPA satisfies the enforced tree-input assumption can be decided in cubic time by trivial reduction to the emptiness problem.

Because VPAs were originally designed for verification and not for XML, the original definition of VPAs is not restricted to trees and hedges. Actually, since we only consider VPA accepting tree languages, we could have used the streaming tree automaton [GNR08, Gau09] formalism for VPAs, except that we do not assume a streaming model for XML. Therefore, an event-oriented formalism would have been slightly more cumbersome (in our setting) than a formalism focusing on nodes, trees and words. The streaming tree automaton model does not use the *enforced tree-input* assumption because the language of an STA is defined under the *expected tree-input* assumption.

We already pointed out that VPAs may sometimes define a set of hedges instead of trees. In particular, given a VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ and a pair of states $q, q' \in Q$, we denote by $\mathcal{A}_{q,q'}$ the VPA obtained from $\mathcal{A}$ by replacing I with $\{q\}$ and F with $\{q'\}$. This VPA accepts a hedge h if and only if $\mathcal{A}$ admits over $lin(h) = a_1 \dots a_m$ a run $(q_0, \sigma_0), (q_1, \sigma_1), \dots (q_m, \sigma_m)$ satisfying $q_0 = q$ and $q_m = q'$ (and $\sigma_0 = \sigma_m = \varepsilon$). We denote by $Acc_{\mathcal{A}}$ the horizontal accessibility relation of $\mathcal{A}$, defined as the set of all pairs (q, q') such that $\mathcal{A}_{q,q'}$ accepts at least one hedge. Formally, $Acc_{\mathcal{A}} = \{(q, q') \in Q^2 \mid L(\mathcal{A}_{q,q'}) \neq \varnothing\}$.

We observe that a run ρ of $\mathcal{A}$ over a tree t induces a function, which we abusively identify with the run ρ, from the nodes of t to pairs of states (q, q'). Given $n \in N_t$, if a_i, a_j is the pair of opening and closing tag corresponding to node n in the word $lin(t) = a_1 \dots a_m$ and ρ is the sequence $(q_0, \sigma_0), (q_1, \sigma_1), \dots (q_m, \sigma_m)$, then $\rho(n)$ is defined as (q_i, q_{j-1}). Note that we have $i = j - 1$ if n is a leaf of t. A run ρ also induces another function $\rho^{\uparrow}$ from nodes to pairs of states, defined by $\rho^{\uparrow}(n) = (q_{i-1}, q_j)$; the states of the VPA before processing the opening tag of n and after processing its closing tag. We point out that $\rho^{\uparrow}$ fails to fully characterize the run as the knowledge of $\rho^{\uparrow}$ for all nodes of the tree is not sufficient to determine the state of the automaton between the opening and closing tag of the leaves. Actually, the pairs $\rho(n)$ for every node n do not characterize the run either as the state between the closing tag of a node and the opening tag of its next-sibling is not represented.

A VPA $\mathcal{A}$ is *deterministic* if $|I| = 1$ and if for every $q \in Q$ and $a \in \Sigma$ the following two conditions are satisfied: (1) for every $\gamma \in \Gamma$, there exists at most one $q' \in Q$ such that $(q, (cl, a), \gamma, q') \in \Delta$, and (2) there exists at most one $\gamma \in \Gamma$ and $q' \in Q$ such that $(q, (op, a), \gamma, q') \in \Delta$. A VPA $\mathcal{A}$ is *unambiguous* if for every word w, $\mathcal{A}$ has at most one accepting run over w. Every deterministic VPA is clearly unambiguous, but the converse does not hold.

Regular Tree Languages A set of unranked trees L is a *regular tree language* if there exists a tree automaton accepting *fcns*(L), or equivalently if there exists a VPA accepting *lin*(L): the visibly pushdown automata over the linearization and the NTA over the *fcns* encoding both have (exactly) the expressive power of *MSO* over trees.

Determinization Tree automata can be determinized, albeit at exponential cost. In particular, given any (binary) tree automaton $\mathcal{A}$, one can compute a deterministic (bottom-up, binary) tree automaton equivalent to $\mathcal{A}$ in time $2^{O(|\mathcal{A}|)}$. Furthermore, this exponential blowup cannot be avoided: there exists a family of languages $L_n, n \geqslant 1$ such that L_n is accepted by a tree automaton of size $O(n)$ but any deterministic bottom-up automaton accepting L_n has 2^n states. The lower bound is an immediate consequence of the blowup for word automata.

Similarly, given any VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, one can compute a deterministic VPA equivalent to $\mathcal{A}$ in time $2^{O(|\mathcal{A}|^2)}$. Furthermore, this complexity involving an exponential with quadratic exponent cannot be avoided: there exists a family of languages $L_n, n \geqslant 1$ such that L_n is accepted by a VPA with $O(n)$ states but any deterministic VPA accepting L_n needs 2^{n^2} states [AM09].

In this dissertation we use the determinization construction to establish the complexity of the evaluation and emptiness problems for VPAs. The construction also underlies the construction of Proposition 5.27 for representing the updates that are not equivalent to updates in a regular language L. We therefore detail below the construction, which is a trivial adaptation for tree languages of the construction from Alur and Madhusudan. This adaptation was already presented in [Gau09, p. 80].

Theorem 3.5 ([AM09]). *Let $\mathcal{A}$ a VPA. One can compute a deterministic VPA $\mathcal{A}'$ satisfying $L(\mathcal{B}) = L(\mathcal{A})$. Moreover, if $\mathcal{A}$ has n states then $\mathcal{B}$ has 2^{n^2} states and 2^{n^2} stack symbols, and can be computed in $O(|\mathcal{A}| \times 2^{2n^2})$.*

Proof. Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA over Σ. We define an automaton $\mathcal{B} = (\Sigma, Q_B, \Gamma_B, I_B, F_B, \Delta_B)$ as follows. The set of states and stack alphabet are both $\mathcal{P}(Q^2)$. The initial state is $I_B = \{\{(q, q) \mid q \in I\}\}$. The set of final states is $F_B = \{S \subseteq Q^2 \mid S \cap (Q \times F) \neq \varnothing\}$. Finally, the transitions are defined below.

For every $S \in \mathcal{P}(Q^2)$ and $a \in \Sigma$, let S' denote the set of all pairs (q, q) such that there exist q_1, q_2 and γ satisfying the following two conditions: (1a) $(q_1, q_2) \in S$, (2a) $q_2 \xrightarrow{(\text{op},a):\gamma} q \in \Delta$. Then Δ_B has transition $S \xrightarrow{(\text{op},a):S} S'$

For every $S, S_0 \in \mathcal{P}(Q^2)$ and $a \in \Sigma$, let S' denote the set of all pairs (q_1, q_5) such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1b) $(q_1, q_2) \in S_0$, (2b) $q_2 \xrightarrow{(\text{op},a):\gamma} q_3 \in \Delta$, (3b) $(q_3, q_4) \in S$, and (4b) $q_4 \xrightarrow{(\text{cl},a):\gamma} q_5 \in \Delta$. Then Δ_B has transition $S \xrightarrow{(\text{cl},a):S_0} S'$.
Let w a word over $\{op, cl\} \times \Sigma$ that is the prefix of the linearization of a tree. Let u the longest well-nested suffix of w, and v the prefix of w before u: $w = vu$. The word v necessarily ends with an opening tag, and u either begins with an opening tag or equals ε if the last symbol of w is an opening tag. The following invariant proves the correction of the construction.

> **Invariant:** *The state reached by VPA $\mathcal{B}$ after reading w is the set of all pairs (q, q') such that u belongs to $L(\mathcal{A}_{q,q'})$ and $\mathcal{A}$ can reach state q after reading v.*

We observe that if w is not the prefix of the linearization of some tree, then clearly the evaluation of $\mathcal{B}$ fails on w. This is because under our *enforced tree-input* assumption, VPAs cannot accept nested words that are not the linearization of trees, so $\mathcal{A}$ by definition would reject w. Since states accessible by $\mathcal{B}$ only contain states accessible by $\mathcal{A}$, $\mathcal{B}$ also rejects w and therefore also satisfies the *enforced tree-input* assumption. But even if we do not require the *enforced tree-input* assumption, we still have $L(\mathcal{B}) = L(\mathcal{A})$ under the *expected tree-input* assumption because of the same invariant. □

There are a few minor differences between this construction and the original one from [AM09]. The main divergence stems from different definitions for VPAs: in our setting, VPAs may only accept tree linearizations. In the model of Alur and Madhusudan, there is no such restriction, and therefore the stack alphabet is $\mathcal{P}(Q^2) \times \Sigma$ (plus a distinct initial stack symbol), the symbol from Σ indicating which opening transitions can be considered in the construction of the closing rules. Since the letters in rules (4b) and (2b) need not be identical in their model, the letter that should be used for rule (2b) has to be stored and recovered from the stack.

There is also an inconsequential difference with the construction in [Gau09]: Gauwin simplifies the opening rule of $\mathcal{A}'$ into $S \xrightarrow{(\mathrm{op},a):S} \{(q,q) \mid q \in Q\}$ instead of $S \xrightarrow{(\mathrm{op},a):S} S'$. The definition of the closing rules as above ensures that both construction are equivalent.

Complexity of determinization The complexity of the construction is in any case dominated by the computation of the closing rules. There are several strategies to compute those rules. Gauwin, for instance, first computes for every possible value of S the set Update_S^a of all pairs q_2, q_5 for which there exist q_3, q_4 and γ satisfying conditions (2b) to (4b) above. Then S' is obtained as the composition of binary relations S_0 and Update_S^a. Let $n = |Q|$ denote the number of states. A rough analysis[3] shows that all relations Update_S^a can be computed in $O(|\Delta| \times n \times 2^{n^2})$. Therefore we claim that we can compute all transitions in time $O(|\Delta| \times n \times 2^{n^2} + |\Sigma| \times n^2 \times 2^{2n^2})$, when the transition table is stored as a set of tuples from $Q_B \times \{op, cl\} \times \Sigma \times \Gamma_B \times Q_B$, with each element from Q_B or Γ_B represented explicitly by a matrix. If we represent the elements Q_B and Γ_B in those tuples by pointers toward the corresponding states instead of matrices, then we can compute the transition table in $O(|\Delta| \times n \times 2^{n^2} + |\Sigma| \times 2^{2n^2})$. With $O(|\mathcal{A}| \times 2^{2n^2})$ we clearly have an upper bound for this complexity.

To establish our claim we need to show how we can compute all closing transitions with these complexities, once the relations Update_S^a are known. We essentially have to compute the composition $S_0 \circ \mathit{Update}_S^a$ for all $a \in \Sigma$ and $S_0, S \in \mathcal{P}(Q^2)$. We can use dynamic programming to compute for any relation $S \subseteq Q^2$ the product of all compositions $S_0 \circ S$ ($S_0 \subseteq Q^2$), in total time $O(n^2 \times 2^{n^2})$. The argument is that there are at most 2^n arrays of n booleans, therefore one can compute the product of all such arrays with R, in total time $n^2 \times 2^n$. Each relation S_0 is represented as a matrix of n such arrays (the lines of the matrix). Consequently the product of S_0 with R can be deduced in time $O(n^2)$ from the precomputed results, hence a total time of $O(n^2 \times 2^{n^2})$ for computing the product of all relations $S_0 \subseteq Q^2$ with R. The claim follows immediately. We shall discuss again the computation of the closing rules for Proposition 3.8.

Disambiguation The transformation of an unambiguous VPA with n states into a deterministic one involves a 2^{n^2} blowup, but so does also the transformation of a VPA into an unambiguous one [OS11]. Consequently, converting VPAs into unambiguous automata instead of deterministic ones does not

[3] see also page 74

guarantee to avoid the exponential blowup.

Closure Properties Regular tree languages are closed under union, intersection, and complementation. Furthermore, given two VPAs $\mathcal{A}_1$ and $\mathcal{A}_2$, we can compute in polynomial time a VPA accepting $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ or $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, and in exponential time a VPA computing $T_\Sigma \backslash L(\mathcal{A}_1)$.

Han and Salomaa [HS09] establish precise bounds for those standard operations. Let $\mathcal{A}_1 = (\Sigma, Q_1, \Gamma_1, I_1, F_1, \Delta_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, \Gamma_2, I_2, F_2, \Delta_2)$ two VPAs. Han and Salomaa give a (standard) construction that builds an automaton for $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ with $|Q_1| + |Q_2|$ states and $\max(|\Gamma_1|, |\Gamma_2|) + 2$ stack symbols, and another construction that builds an automaton for $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ with $|Q_1| \times |Q_2|$ states and $|\Gamma_1| \times |\Gamma_2|$ stack symbols. They show that those results are optimal in the following sense: for every n, there exist automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with n states and n stack symbols, such that every automaton accepting $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ has at least n^2 states and n^2 stack symbols. Similarly, for every n, there exists automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with n states and n stack symbols, such that every automaton accepting $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ has at least $3n + 2$ states plus stack symbols.

The bounds for complementation, have been tightened more recently: for every VPA $\mathcal{A}$, one can build an automaton of size $2^{O(|\mathcal{A}|^2)}$ that accepts the complement of $L(\mathcal{A})$ [AM09]. For arbitrarily large n there exists a VPA $\mathcal{A}_n$ with n states and stack symbols such that any VPA accepting the complement of $L(\mathcal{A})$ has $2^{(\Omega(|\mathcal{A}|^2))}$ states plus stack symbols [OS11]. For a more detailed analysis of all these state complexities, we refer the reader to the results in [HS09] and [OS11], which are slightly more precise.

Alternative Automata Models for Unranked Trees

Several other formalisms have been proposed for automata over unranked trees [CDG+07]. Binary tree automata over the curryfied encoding have been considered, for instance, in [CNT04], where they are called *stepwise automata*. *Hedge automata* form another popular model working directly on the unranked tree, without further encoding: transitions are of the form (q, a, L) where L is a regular language describing the states on the children of the node. Hedge automata have received much attention since the early works on unranked tree languages [Mur99, BKMW01]. Several automata models based on hedge automata are presented in [BKMW01], with an overview of related historical results on regular tree languages for ranked and unranked trees.

Alternating Automata, and Tree-Walking Automata Intuitively, all the models defined above traverse each node of the tree at most twice, and their semantics is easily defined by the notion of runs, i.e., functions mapping in a straightforward way each node of the tree to a state (or a pair of states).

Tree-walking automata and two-way-alternating automata follow a different philosophy, as their execution does not follow a particular traversal of the tree, but can examine several times the same node in the tree. The "execution" of a tree-walking automaton over a tree is purely sequential, whereas two-way alternating automata make a heavy use of branching.

Tree-walking automata were introduced in [AU71]. They have been used recently in order to prove the gaps in terms of expressiveness between the logics *MSO* and Regular XPath [tCS10] (see also page 93). Bojańczyk surveys in [Boj08] properties of tree-walking automata, focusing on expressiveness of the different types of tree-walking automata. The languages accepted by (standard) tree-walking automata are clearly regular tree languages, but the question whether the other inclusion holds was not solved until 2008, when Bojańczyk and Colcombet produced a proof that tree-walking automata are strictly less expressive than *MSO* [BC08]. There exists a notion of determinism for tree-walking automata but deterministic tree-walking automata are strictly less expressive than (non-deterministic) tree-walking automata [BC06]. Because of these limitations we do not use tree-walking automata in this dissertation, notwithstanding their strong connection to Regular XPath [tC06], but use the more expressive two-way alternating automata instead. For some XPath fragments, however, algorithms testing emptiness of tree-walking automata could prove a viable alternative to emptiness testing for 2-ATAs. Héam et al. [HHK11] made a further step in that direction, through the investigation of translations from tree-walking automata to NTAs over full binary trees. Their algorithm focus on the computations of the possible "loops" of the tree-walking automaton, and yield efficient algorithms, especially for deterministic tree-walking automata. The authors also propose approximations to bypass the Exptime-completeness of emptiness checking for tree-walking automata.

In an NTA, the transition function maps each pair (q, a) to a set S of pairs q_1, q_2. The pair q_1, q_2 expresses a kind of conjunction: there must be an accepting run from q_1 in the left child of the node *and* an accepting run from q_2 in the right child of the node. Intuitively, the set S expresses a disjunction: one can choose which pair of states will be attributed to the children. Alternating automata replace this set of pairs (or equivalently, this disjunction of conjunctions) by an arbitrary positive Boolean formula to specify the properties that must be satisfied by the children of the node. Two way alternating automata allow the formula to express conditions about the parent node in addition to the conditions on the children nodes. *Two-way (weak) alternating automata* (2-ATAs) were introduced in [CGLV09] as a specialization for finite tree of the two-way weak alternating automata from [KVW00]. It is argued in [CGLV09, CGLV10] that 2-ATAs provide an interesting model for reasoning about Regular XPath. In particular, they provide a hopefully more implementable alternative to reasoning about variants of Propositional Dynamic Logic (PDL) that also have Exptime-complete satisfiability, but for which the exponential satisfiability algorithm is hardly implementable due to the use of the notoriously complex determinization procedure from Safra. The construction in [CGLV09, CGLV10] converts Regular XPath formulae in linear time into a 2-ATA, and then builds an NTA equivalent to this 2-ATA in exponential time. The conversion is discussed further in subsection 3.3.6. We also postpone the formal definition of 2-ATA to this subsection. We nonetheless survey here the complexity of the usual decision problems for 2-ATA. Satisfiability is EXPTIME-hard in general since 2-ATAs extend NTAs, and is therefore EXPTIME-complete, but we prove that it becomes PSPACE-complete over (binary encoding of) a non-recursive DTD, i.e., when considering only binary trees that encode trees of depth polynomial in the size of the automaton. We do not write the proof explicitly, but it can be obtained immediately from the proof of proposition 4.31, taking 2-ATAs instead of Regular XPath formulae. As observed in [CGLV09], given a 2-ATA $\mathcal{A}$ and a tree t, one can decide in time $O(|\mathcal{A}| \times |t|)$ if t belongs to $L(\mathcal{A})$ using the technique from [KVW00].

The *fcns* encoding sometimes provides too little information for the simulation of XPath formulae with tree-walking or two-way automata, in particular one cannot infer whether the current node is the first child of its parent node or the second when using the parent axis of the binary tree: a transition going to the parent node in $fcns(t)$ might correspond in the unranked tree to either a move to the parent, or a move to the previous sibling. To remedy this shortcoming of the *fcns* encoding, one can enrich the label of each node with a subset of $\{ifc, irs, hfc, hrs\}$. Thus the new labeling function maps each node to some element in $\Sigma \times \mathcal{P}(\{ifc, irs, hfc, hrs\})$. The symbols ifc, irs, hfc, hrs respectively stand for “is first child”, “is right sibling”, “has first child”, and “has right sibling”. For instance, in the decorated binary encoding t' of an unranked tree t, a node label contains *ifc* iff the node is the leftmost child of some node in t, and the nodes of t' whose label contains *hrs* have a right sibling in t, hence have a right child in t' with label $a \in \Sigma, a \neq \perp$, etc.

The above solution follows the presentation of [CGLV09]. Bojańczyk and Parys [BP11] adopt a very similar approach, as they distinguish two kinds of parent axes for the navigation over the binary tree: `from-left` and `from-right` that can be applied only from the left and right child of a node respectively. Ten Cate and Lutz [tCL09] follow an approach similar to Bojańczyk and Parys. Although they do not explicitly mention a binary encoding, their 2-ATA traverse trees using the axes corresponding to the *fcns* structure, one of the axes goes to the previous sibling in the unranked tree, and another goes to the parent provided the current node has no left sibling, which corresponds to the `from-right` and `from-right` axes over the *fcns* encoding, respectively.

Equivalence of Tree Automata Models up to Polynomial Conversion

All those models have the same expressiveness. Furthermore, they are equivalent up to a polynomial translation if we assume that the regular languages in the transition of the hedge automata are given by NFAs in the case of hedge automata. Let us detail the transformations between VPAs and tree automata over the *fcns* encoding, as presented in [Gau09].

From NTAs to VPAs Let $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ be an NTA. The VPA $\mathcal{A}' = (\Sigma, Q', \Gamma', I', F', \Delta')$ defined as follows satisfies $\mathit{fcns}(L(\mathcal{A}')) = L(\mathcal{A})$. We pose $Q' = Q$, $\Gamma' = Q \times \Sigma$, $I' = Q_f$, and $F' = Q$. The transitions of $\mathcal{A}'$ are derived from those of $\mathcal{A}$: $\mathcal{A}'$ has transition $q \xrightarrow{(\mathrm{op},a):(q_2,a)} q_1$ for every transition $a(q_1, q_2) \to q$ in Δ, and has transition $q \xrightarrow{(\mathrm{cl},a):(q_2,a)} q_2$ for every transition $\perp \to q$ in Δ and every $q_2 \in Q$. This translation preserves unambiguity, but it does not preserve bottom-up determinism, that is, even when $\mathcal{A}$ is a bottom-up deterministic automaton, $\mathcal{A}'$ needs not be a deterministic VPA.[4] The conversion above yields a VPA satisfying the strong enforced tree-input assumption. Under the expected input-tree assumption, the conversion can be simplified by dropping the Σ component from Γ'.

Example 3.4. *Let $\mathcal{A}$ be the NTA with two states q_0, q_1 and with transitions $\perp \to q_0$, $a(q_0, q_1) \to q_0$, $a(q_0, q_1) \to q_1$, $b(q_0, q_0) \to q_0$, and $b(q_0, q_0) \to q_1$. We set to $\{q_0\}$ the final states of $\mathcal{A}$. The NTA $\mathcal{A}$ accepts the fcns encoding of the trees over $\Sigma = \{a, b\}$ in which the rightmost child of each node has label b. In other words, $\mathcal{A}$ accepts all binary trees over $\{a, b, \perp\}$ satisfying the two conditions that (1) all and only the leaves are labeled $\perp$, and (2) each node whose right child is a leaf has label b. Figure 3.4 represents the VPA obtained from $\mathcal{A}$ while Figure 3.5 parallels the runs of $\mathcal{A}'$ and $\mathcal{A}$ over tree $t = b(a(b), a, b)$ and its fcns encoding. In both figures we have dropped the Σ component from the stack symbols for better readability. This examples thus illustrates the conversion under the expected input-tree assumption.*

In the run of $\mathcal{A}$ over fcns(t), the transition $a(q_0, q_1) \to q_0$ is applied at the left child of the root. Consequently, $\mathcal{A}'$ remains in state q_0 and pushes q_1 (resp. (q_1, a) for the conversion under enforced tree-input assumption) onto the stack when it processes the opening tag of this node in t. The state q_1 (resp. (q_1, a)) is therefore popped when processing the closing tag, so that q_1 is the state of $\mathcal{A}'$ before processing the opening tag of the other a node. More generally we can check the state of $\mathcal{A}'$ before processing the opening tag of a node n in t is the state assigned to n by $\mathcal{A}$ in fcns(t). We observe also that $\mathcal{A}'$ is a deterministic VPA, which was to be expected since $\mathcal{A}$ is top-down

[4] When $\mathcal{A}$ is a top-down deterministic automaton, however, the VPA $\mathcal{A}'$ obtained by the conversion is deterministic [Gau09], but we do not study top-down determinism in this dissertation.

$$
\begin{aligned}
&\bot \rightarrow q_0,\\
&a(q_0, q_1) \rightarrow q_0,\\
&a(q_0, q_1) \rightarrow q_1,\\
&b(q_0, q_0) \rightarrow q_0,\\
&b(q_0, q_0) \rightarrow q_1
\end{aligned}
$$

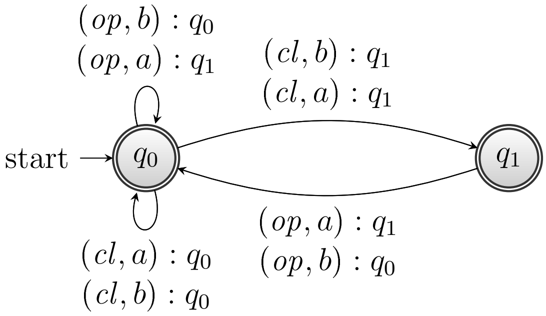
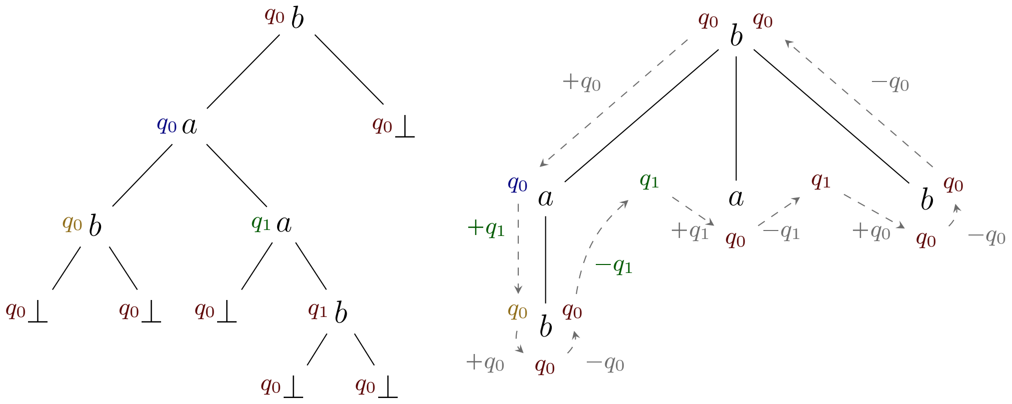


Figure 3.4.: NTA $\mathcal{A}$ and VPA $\mathcal{A}'$



run of NTA $\mathcal{A}$ over $fcns(t)$ run ρ_t of VPA $\mathcal{A}'$ over tree t

Figure 3.5.: From NTAs to VPAs

deterministic. We also observe that transitions $(cl, a) : q_0$ and $(cl, b) : q_1$ are useless in $\mathcal{A}'$.

Proposition 3.6. *Given any NTA $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ one can build in linear time $O(|\Delta|)$ a VPA $\mathcal{A}'$ such that $fcns(L(\mathcal{A}')) = L(\mathcal{A})$. The VPA $\mathcal{A}' = (\Sigma, Q', \Gamma', I', F', \Delta')$, computed by this conversion satisfies $|Q'| = O(|Q|)$, $|\Gamma'| = O(|\Gamma| \times |\Sigma|)$, and $|\Delta'| = O(|\Delta|)$.*

Proof. The conversion detailed above almost satisfies those requirements except for the closing rules that may be too many. If however we make sure there is a unique state q such that $\bot \rightarrow q$ belongs to Δ, then $\mathcal{A}'$ has only $O(|Q|)$ closing rules. This can easily be achieved if we introduce a new state q' in Q and add to Δ the transition $\bot \rightarrow q'$ together with appropriate transitions $a(q, q_2) \rightarrow q_0$, $a(q_1, q) \rightarrow q_0$ and $a(q, q) \rightarrow q_0$, for a total of $O(|\Delta|)$ transitions. □

From VPAs to NTAs Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA. The NTA $\mathcal{A}' = (\Sigma, Q', Q'_f, \Delta')$ defined as follows satisfies $\mathit{fcns}(L(\mathcal{A})) = L(\mathcal{A}')$. We pose $Q' = Q \times Q$, $Q'_f = I \times F$. For every $a \in \Sigma$, $\mathcal{A}'$ has transition $a((q_1, q_2), (q_3, q_4)) \to (q_0, q_4)$ if and only if there exists $\gamma \in \Gamma$ such that $q_0 \xrightarrow{(\mathrm{op},a):\gamma} q_1$ and $q_2 \xrightarrow{(\mathrm{cl},a):\gamma} q_3$. $\mathcal{A}'$ has transition $\bot \to (q, q)$ for every $q \in Q$. Again, this translation preserves unambiguity, but it does not preserve determinism. The number of states in $\mathcal{A}'$ may be quadratic in the number of states in $\mathcal{A}$, and even in the size of $\mathcal{A}$. We prove below that this cannot be helped in general. The number of transitions in $\mathcal{A}'$ may exceed $|\mathcal{A}|^2$ but is bounded by $|Q| \times |\Delta|^2$.

Proposition 3.7. *Given any VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, one can build in time $O(|Q| \times |\Delta|^2)$ an NTA $\mathcal{A}'$ such that $\mathit{fcns}(L(\mathcal{A})) = L(\mathcal{A}')$. The NTA $\mathcal{A}'$ computed by this conversion has $O(|Q|^2)$ states and $O(|Q| \times |\Delta|^2)$ transitions.*

For every $n \in \mathbb{N}$ there exists a deterministic VPA $\mathcal{A}_n$ of size $O(n)$ over a unary alphabet and using a single stack symbol, such that any NTA equivalent to $\mathcal{A}_n$ needs at least n^2 states.

Proof. Let $n \in \mathbb{N}$. Let $\mathcal{A}$ the VPA defined by $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, with $Q = \{q_1, \ldots, q_n, p_1, \ldots, p_{n-1}\}$, $\Gamma = \{\gamma\}$, $\Sigma = \{a\}$, $I = \{q_n\}$, $F = \{p_1\}$, and with transitions defined as follows:

$$\begin{aligned}\Delta = &\{q_i \xrightarrow{(\mathrm{op},a):\gamma} q_{i+1} \mid i < n\} \cup \{q_n \xrightarrow{(\mathrm{op},a):\gamma} q_1\} \cup \{q_n \xrightarrow{(\mathrm{cl},a):\gamma} p_{n-1}\} \\ &\cup \{p_i \xrightarrow{(\mathrm{cl},a):\gamma} p_{i-1} \mid i > 1\} \cup \{p_1 \xrightarrow{(\mathrm{cl},a):\gamma} p_{n-1}\}\end{aligned}$$

The VPA $\mathcal{A}$ only accepts unary trees. Intuitively, the states q_i check that the number of opening tags is a multiple of n, while the p_j check the number of closing tags is a multiple of $n-1$. Consequently, a unary tree is accepted by $\mathcal{A}$ if and only if its depth is a multiple of $\mathit{gcd}(n, n-1) = n(n-1)$. The VPA $\mathcal{A}$ has size $O(n)$, but the usual pumping argument for (ranked) tree automata implies that any NTA accepting $L(\mathcal{A})$ needs $n(n-1)$ states. □

With a second stack symbol γ' one can easily modify $\mathcal{A}$ so that it accepts only the unary tree of depth $n(n-1)$: we replace transition $q_{n-1} \xrightarrow{(\mathrm{op},a):\gamma} q_n$ by $q_{n-1} \xrightarrow{(\mathrm{op},a):\gamma'} q_n$, replace transition $q_n \xrightarrow{(\mathrm{cl},a):\gamma} p_{n-1}$ by $q_n \xrightarrow{(\mathrm{cl},a):\gamma'} p_{n-1}$, and add a transition $p_i \xrightarrow{(\mathrm{cl},a):\gamma'} p_{i-1}$ for each $i > 1$. We also observe that the construction can be simplified if we allow a quadratic number of stack symbol.

Figure 3.6 summarizes the cost of conversions between our tree automata models in terms of number of states and number of transitions. We do not consider the number of transitions for alternating automata because the transition rules have a different nature: we do not want to compare boolean formula with set of tuples.

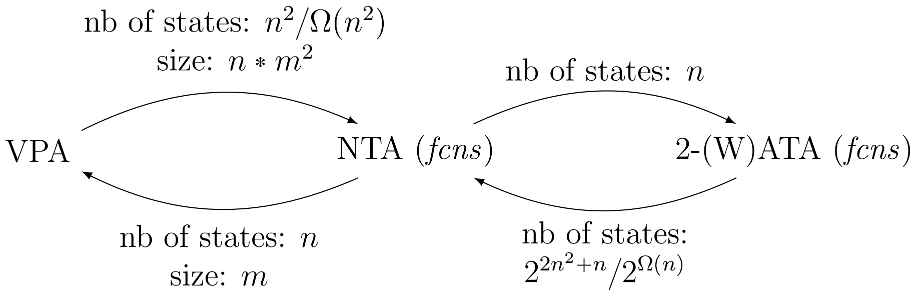


Figure 3.6.: Number of states and size obtained from the conversion of an automaton with n states and m transitions

3.2.2. Decision Problems for Tree Automata

For tree automata as well as for VPAs, the containment and the equivalence problems are EXPTIME-complete. Emptiness and Membership can be tested in polynomial time. Complexities for emptiness and membership are classical results for tree automata models, so we only survey those that will prove useful in this dissertation. Emptiness can be decided in linear time for a ranked tree automaton, by reduction to CFG emptiness for instance: each transition rule corresponds to one possible step of a derivation. The emptiness and membership problem are clearly decidable in polynomial time for visibly pushdown automata, but the polynomials involved have received scant attention in the literature as far as we could observe. The next few paragraphs are therefore devoted to the complexity of emptiness and membership for visibly pushdown automata.

Emptiness for VPAs We survey several approaches to decide emptiness for VPAs, namely:

1. the reduction to ranked tree automata emptiness through the conversion from Proposition 3.7
2. the reduction to CFG emptiness
3. the computation of an NFA representing all reachable configurations
4. the computation of the horizontal reachability relation of the VPA $\mathcal{A}$

We obtain the following complexity result:

Proposition 3.8. *Given a VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, one can compute the horizontal reachability relation of $\mathcal{A}$, and therefore decide emptiness of $L(\mathcal{A})$ in $O(|\Delta| \times |Q| + |Q|^3)$.*

Remark 3.1. *The $O(|Q|^3)$ term comes from computing the transitive closure of $Acc_{\mathcal{A}}$, and matrix-multiplication techniques allow to compute transitive closure of a relation over Q with complexity $O(|Q|^{\omega})$. But the transitive closure rules are applied incrementally simultaneously with other rules, so it is not clear whether the matrix techniques could be of any help here. In any case, the $O(|\Delta| \times |Q|)$ term probably subsumes the $O(|Q|^3)$ contribution in most practical automata.*

The conversion to NTA from Proposition 3.7 provides a first algorithm to decide emptiness of visibly pushdown automata. The resulting NTA, however, has size $O(|Q| \times |\Delta|^2)$ so that applying the standard linear algorithm to decide emptiness of the resulting NTA without exploiting the specific structure of the resulting NTA is not optimal: the overall complexity in that case would be $O(|Q| \times |\Delta|^2)$, which is greater than $O(|\Delta|^2 + |Q|^3)$.

The conversion of pushdown automata into context-free grammars is a classical solution to test emptiness of pushdown automata. The standard construction for pushdown automata builds from $\mathcal{A}$ the CFG $G = (V, T, S, P)$ defined by a set of non terminals $V = \{S\} \cup Q \times (\Gamma \cup \{\varepsilon\}) \times Q$, delimited by brackets [ ], and terminals $T = \{op, cl\} \times \Sigma$. The productions P consist of (1) one production $[q, \gamma, q'] \to (cl, b)$ for every $q \xrightarrow{(\mathrm{cl},b):\gamma} q' \in \Delta$, (2) one production $[q, \gamma, q''] \to (op, b)\,[q', \gamma', q'''][q''', \gamma, q'']$ for every $q \xrightarrow{(\mathrm{op},b):\gamma'} q' \in \Delta$, $\gamma \in \Gamma \cup \{\varepsilon\}$ and $q'', q''' \in Q$, (3) one production $[q, \varepsilon, q] \to \varepsilon$ for every $q \in Q$, and (4) one production $S \to [q', \gamma, q'']$ for each $q \in I$, $q' \in Q$, and $q'' \in F$ such that $q \xrightarrow{(\mathrm{op},a):\gamma} q' \in \Delta$. Consequently, G may contain $|\Delta| \times |Q|^2 \times |\Gamma|$ productions, and this approach yields an algorithm in $O(|\Delta| \times |Q|^2 \times |\Gamma|)$ to decide the emptiness problem for $\mathcal{A}$. This straightforward reduction to CFG emptiness is therefore less efficient than the next algorithms which present $O(|Q|^3 \times |\Gamma| + |\Delta|)$ worst case complexity.

A third approach, popular in the model-checking community, relies on the observation that for any pushdown automaton, the set of all configurations reachable from the initial states is regular. Moreover an NFA representing this set can be computed from $\mathcal{A}$ in cubic time [FWW97]. Using this NFA one can easily check whether there exists some final state that can be reached with an empty stack.

This result is often presented in a broader setting: for every regular set of configurations C, one can compute an automaton for the set $Post^*(C)$ of configurations that can be reached from some configuration in C. Similarly, one can compute an automaton for the set $Pre^*(C)$ of configurations from which one can reach some configuration in C [BEM97, EHRS00]. The complexities mentioned in those works depend on the formalism assumed for the pushdown automaton and so in general need to be slightly adapted for visibly pushdown automata. Van Tang [Tan09] reformulates the construction of the reachable configurations $Post^*(C)$

for visibly pushdown automata.

We analyze next the complexity of computing the configurations reachable from I. We define an NFA $A_{\mathcal{P}}$ with initial states Q, such that a word $w \in \Gamma^*$ is accepted by $A_{\mathcal{P}}$ from q if and only if the configuration (q, w) is accessible in $\mathcal{A}$ from some configuration (q_i, ε) with $q_i \in I$.[5]

Formally, let $A_{\mathcal{P}} = (\Gamma, Q, Q, I, \Delta_{\mathcal{P}})$ the NFA over alphabet Γ, and with transitions $\Delta_{\mathcal{P}}$ defined according to the following saturation rules:

1. we put in $\Delta_{\mathcal{P}}$ a rule of the form $q' \xrightarrow{\gamma} q$ for every rule $q \xrightarrow{(\mathrm{op},a):\gamma} q'$ in Δ
2. we put in $\Delta_{\mathcal{P}}$ a rule of the form $q''' \xrightarrow{\epsilon} q$ for every pair of rules $q \xrightarrow{(\mathrm{op},a):\gamma} q'$ and $q'' \xrightarrow{(\mathrm{cl},b):\gamma} q'''$ in Δ such that $q''(\xrightarrow{\epsilon})^* q'$ according to $\Delta_{\mathcal{P}}$.

We observe that in the second rule, a and b should actually refer to the same letter according to our restriction of VPAs to linearizations of trees, but we keep different letters in order to cover the more general case. Our construction differs in many respects from the one in [Tan09], essentially because of a different definition of VPA and a simpler setting for the problem. Indeed Van Tang details the construction of $Post^*(C)$ for an arbitrary regular set of configurations C, and his definition of VPAs both assumes an initial stack symbol and allows internal transitions. The construction that computes the reachable configurations corresponds to the rules for computing the horizontal reachability relation. Indeed, q' can be reached from q following ϵ-transitions in Δ_p if and only if $(q, q') \in Acc_{\mathcal{A}}$. This construction can therefore be implemented with complexity $O(|\Delta| \times |Q|)$, as we shall discuss in the next paragraph.

A fourth method to solve the emptiness problem is to compute the horizontal accessibility relation $Acc_{\mathcal{A}}$. Clearly, the horizontal accessibility relation of $\mathcal{A}$ is the smallest subset of Q^2 satisfying the following three conditions:

1. $\{(q,q) \mid q \in Q\} \subseteq Acc_{\mathcal{A}}$
2. $Acc_{\mathcal{A}}$ is closed under transitive closure: if $(q, q') \in Acc_{\mathcal{A}}$ and $(q', q'') \in Acc_{\mathcal{A}}$ then $(q, q'') \in Acc_{\mathcal{A}}$.
3. for every transitions $q \xrightarrow{(\mathrm{op},a):\gamma} q'$ and $q'' \xrightarrow{(\mathrm{cl},a):\gamma} q'''$ in Δ, if $(q', q'') \in Acc_{\mathcal{A}}$ then $(q, q''') \in Acc_{\mathcal{A}}$

This relation can therefore be computed with complexity $O(|\Delta|^2 + |Q|^3)$, as observed in [Gau09, p. 108]. Actually, Gauwin does not state the result as such: he mentions the result in a particular setting, and for deterministic VPAs only, but his proof does not require determinism. With a careful attention to the strategy (order of the operations) to compute the relation, we can actually obtain $Acc_{\mathcal{A}}$ in $O(|\Delta| \times |Q| + |Q|^3)$.

[5] This NFA is often called the $\mathcal{P}$-automaton of $\mathcal{A}$ in the model-checking literature.

To reduce the $|\Delta|^2$ term to $O(|\Delta| \times |Q|)$ we compute the relation $R_1 = \{(q, a, \gamma, q'') \mid q \xrightarrow{(\mathrm{op},a):\gamma} q' \in \Delta\}$ together with $Acc_{\mathcal{A}}$: each time a new pair (q', q'') is added to $Acc_{\mathcal{A}}$ we add all corresponding tuples (q, a, γ, q'') into R_1. And whenever a new tuple (q, a, γ, q'') is added into R_1, we add to $Acc_{\mathcal{A}}$ the corresponding pairs (q, q''') generated by rule 3. Of course whenever a pair is added to $Acc_{\mathcal{A}}$ we also take care to add the pairs generated by rule 2. The pairs generated by rule 2 contribute the $|Q|^3$ factor, while the computation of R_1 and the applications of rule 3 contribute the $|\Delta| \times |Q|$ factor. We also observe that under the enforced tree-input hypothesis one can project out the Σ component from Δ before we apply the above construction.

The emptiness of $L(\mathcal{A})$ can be established using the accessibility relation: $L(\mathcal{A}) \neq \emptyset$ if and only if $Acc_{\mathcal{A}} \cap I \times F \neq \emptyset$. This concludes our survey of techniques to decide emptiness for visibly pushdown automata.

In a nutshell, the cost of converting $\mathcal{A}$ into an equivalent ranked tree automaton or context-free grammar via the standard techniques exceeds the cost of directly checking emptiness. The construction of an automaton that accepts the reachable configurations and the computation of the horizontal accessibility relation both provide an algorithm with complexity $O(|\Delta| \times |Q| + |Q|^3)$.

Membership for VPAs We may consider several approaches to decide membership for VPAs, namely:

1. the reduction to the membership problem for ranked tree automata through the conversion from Proposition 3.7
2. the reduction to CFG parsing
3. the determinization of $\mathcal{A}$
4. the determinization of $\mathcal{A}$ “on-the-fly”

Proposition 3.9. *Given a tree t and VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, one can decide if $\mathcal{A}$ accepts t with any of the following complexities, depending on the evaluation strategy adopted.*

- $O(|\mathcal{A}|^2 \times 2^{2Q^2} + |t|)$,
- $O((|\Delta| \times |Q| + |Q|^3) \times |t|)$,
- *or* $O(|\mathcal{A}| \times 2^{2|Q|} + (|Q|^2 \times |\Gamma|) \times |t|)$.

Remark 3.2. *Actually, the Δ contribution in the second bound could be replaced with $\delta = \max_{a \in \Sigma} \Delta \cap (Q \times \{op, cl\} \times \{a\} \times \Gamma \times Q)$, provided one also adds $|\mathcal{A}|$ or $|\Delta|$ in the complexity to make sure the whole input can be read at least once, which gives $O(|\mathcal{A}| + (\delta \times |Q| + |Q|^3) \times |t|)$. In terms of $|Q|$ and*

$|\Gamma|$, δ is therefore bounded by $2 \times |Q|^2 \times |\Gamma|$, so that using fast matrix multiplication, one can replace the bound with any of $O((|\Delta| \times |Q| + |Q|^{\omega}) \times |t|)$ or $O(|\mathcal{A}| + |Q|^{\omega} \times |\Gamma| \times |t|)$.

The cost of evaluating the VPA $\mathcal{A}$ through conversion into an equivalent ranked tree automaton or context-free grammar via the standard techniques exceeds the cost of our evaluation algorithms. We will therefore only present our algorithms working directly on the VPA model. Theorem 3.5 provides an algorithm to evaluate VPA $\mathcal{A}$ over a tree t in time $O(|\mathcal{A}|^2 \times 2^{2|Q|^2} + |t|)$: the evaluation of a deterministic automaton over the tree takes linear time, and the deterministic automaton can be computed in time $O(|\mathcal{A}|^2 \times 2^{2|Q|^2})$ (and even slightly faster). The huge exponent may be prohibitive for a large VPA, all the more so as the storage of the deterministic automaton essentially requires this amount of space (there may be up to $2^{|Q|^2}$ different states).

The determinization of $\mathcal{A}$ *on-the-fly* trades storage space for processing time at evaluation. In a first step (A0) we preprocess in linear time $O(|\Delta|)$ the transition function so that given any $(q, a) \in Q \times \Sigma$, one can obtain in time $O(|Q|)$ the list of all states q' for which there exists some γ such that $q \xrightarrow{(\text{op},a):\gamma} q' \in \Delta$. This preprocessing can be achieved in $O(|\Delta|)$, using lazy arrays [MS90] for instance. We also assume the states $S \subseteq Q^2$ to be stored as arrays of dimension Q^2 (A1) together with a structure that allows to test in constant time for every $q_2 \in Q$ if there exists any q_1 such that (q_1, q_2) belongs to the current state (A2).

When processing an opening tag, from the state $S \subseteq Q^2$ and given a letter $a \in \Sigma$, one has to compute the set S' satisfying conditions discussed below Theorem 3.5, namely the set of all pairs (q, q) such that there exist q_1, q_2 and γ satisfying the following two conditions: (1a) $(q_1, q_2) \in S$, (2a) $q_2 \xrightarrow{(\text{op},a):\gamma} q \in \Delta$. Thus, opening transitions can be processed with complexity $O(\delta \times |Q|)$ according to assumptions (A0) and (A2), or even $O(|Q|^{\omega} \times |\Gamma|)$.

The closing tags account for most of the work. At a closing tag, one gets two sets S and $S_0 \in \mathcal{P}(Q^2)$, a letter $a \in \Sigma$, and one has to compute the set S' of all pairs (q_1, q_5) such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1) $(q_1, q_2) \in S_0$, (2) $q_2 \xrightarrow{(\text{op},a):\gamma} q_3 \in \Delta$, (3) $(q_3, q_4) \in S$, and (4) $q_4 \xrightarrow{(\text{cl},a):\gamma} q_5 \in \Delta$. The challenge is therefore similar to the problem of applying the saturation rule 3 when computing $Acc_{\mathcal{A}}$ for checking emptiness of a VPA as discussed on page 74. We first compute the set S'' of pairs (q_2, q_5) satisfying conditions (2) to (5): the pairs (q_1, q_5) satisfying conditions (1) to (5) are easily deduced as the composition of two binary relations. This composition can be computed easily in $O(|Q|^3)$ or even $O(|Q|^{\omega})$ using fast matrix multiplication, according to Lemma 3.2 and Lemma 3.1.

We first compute the set R_1 of triples (q_2, γ, q_4) for which there exists q_3 such that conditions (2) and (3) are satisfied. This relation R_1 can be computed in $O(\delta \times |Q|)$ according to Lemma 3.2, or even $O(|Q|^{\omega} \times |\Gamma|)$

using fast matrix multiplication. From R_1, we similarly obtain the set S'' in $O(\delta \times |Q|)$, or again $O(|Q|^\omega \times |\Gamma|)$. This approach yields the overall bound of $O((\delta \times |Q| + |Q|^\omega) \times |t|)$ for the membership problem.

We recall that in our variant of the determinization construction, S may only consist of pairs (q_3, q_4) for which there exist q_1 and q_2 and γ satisfying $(q_1, q_2) \in S_0$ and $q_2 \xrightarrow{(\mathrm{op},a):\gamma} q_3 \in \Delta$. The simpler variant from [Gau09] described below the determinization construction drops this assumption, but this has no consequence on the complexities above since the cost of opening transitions is dominated by the cost of closing transitions. The assumption may only help to decrease the size of the state, which may slightly simplify evaluation. However, some other optimization techniques can be applied more easily when we drop the assumption. We recall (from the invariant of Theorem 3.5) that, if we denote by $\mathcal{B}$ the determinized VPA obtained from $\mathcal{A}$, the state reached by $\mathcal{B}$ after reading a well-nested word u is the set of all pairs (q, q') such that $u \in L(\mathcal{A}_{q,q'})$. Let us consider a well-nested word u as a sequence of trees with roots $n_1, \ldots, n_k$. and set $J_1, \ldots, J_k \subseteq Q^2$ such that for each $i \leqslant k$, $J_i = \{(q, q') \mid t_i \in L(\mathcal{A}_{q,q'})$. Then the state reached by $\mathcal{B}$ after closing n_k is $J_1 \circ J_2 \circ \cdots \circ J_k$. The strategies defined above would compute the composition in left-to-right order. This is a requirement for streaming evaluation as considered in [Gau09], but when a streaming evaluation is not required, it may be beneficial to optimize the computation of such compositions, all the more so since statistics [BMV06] show that XML trees generally have low depth, but may have very high arity. One possible direction for the optimization would be to take into account the cardinality of the relations. Another one may be to use parallelization. A last one may be to consider the computation of $J_1 \circ J_2 \circ \cdots \circ J_k$ as a reachability problem over the graph whose vertices are $k+1$ copies $q_1, \ldots, q_{k+1}$ of each state q in Q, the edges between the i^{th} and $(i+1)^{\text{th}}$ copies being given by J_i: one must compute for each $q' \in Q$ the states q_{k+1} reachable from ${q'}_0$. This provides a solution with complexity $O(|Q| \times (|J_1| + |J_2| + \cdots + |J_k|))$, which may sometimes be better than algorithms computing the compositions pair after pair. When k is much larger than $|Q|$, one possibility to speedup the evaluation of the compositions could be to precompute for all pairs of binary relations over $|Q|$ elements the result of their composition, and store this in some table. There are $2^{2|Q|^2}$ possible pairs of such relations, however, so we fall back to the complexity and huge space requirements of determinization. This could be expected since this precomputation is actually the most expensive part in the determinization procedure. We next present a kind of hybrid method that achieves another tradeoff between storage space and processing time. The method also provides a solution to compute $J_1 \circ J_2 \circ \cdots \circ J_k$ in $O(|Q| \times 2^{2|Q|} + |Q|^2 \times k)$.

One can compute in time $O(|Q| \times 2^{2|Q|} + |\Sigma| \times |Q|^3 \times |\Gamma| \times 2^{|Q|}))$ a representation of the determinized VPA that allows to simulate each transition of the deterministic VPA in time $O(|Q|^3)$ (or alternatively $O(|Q|^2 \times |\Gamma|)$ per transition). The constructions rely on a dynamic programming scheme extending the one we presented for the determinization procedure: we precompute the composition of every pair of binary relation over Q. But instead of storing explicitly the result as a matrix, which would require to store up to $2^{|Q|^2}$ different matrices, we only store a representation of the matrix, which allows to compute the matrix in time $O(|Q|^2)$: in a nutshell we add one level of indirection to spare space and time during the preprocessing. We next describe our algorithm to compute a representation of $S \circ R$ for every $S, R \subseteq Q^2$. We observe there are at most $2^{|Q|}$ different arrays of $|Q|$ booleans. Let $\vec{R}$ be the matrix with dimensions $2^{|Q|} \times |Q|$ whose lines are formed of distinct arrays of ($|Q|$) booleans, sorted lexicographically. Let $\vec{S}$ be the transpose of $\vec{R}$: $\vec{S}$ consists of $2^{|Q|}$ distinct arrays of booleans, stored lexicographically in columns.

We can compute the product M of $\vec{R}$ with $\vec{S}$ in $O(|Q| \times 2^{2|Q|})$. The composition of two relations $R, S \subseteq Q^2$ can be deduced from M in $O(|Q|^2)$: if R is formed of the lines $i_1, i_2, \ldots, i_{|Q|} \in \{1, \ldots, 2^{|Q|}\}$, and S of the columns $j_1, j_2, \ldots, j_{|Q|} \in \{1, \ldots, 2^{|Q|}\}$, then $R \circ S$ is formed by the intersection of those lines and columns in M. This justifies that we can compose relations within the complexity claimed. It remains to prove that the relations Update^a_S can also be computed efficiently.

Let $\gamma_1, \ldots, \gamma_{|\Gamma|}$ denote the symbols of Γ in some fixed order. For each $a \in \Sigma$ and $i \leqslant |\Gamma|$, we also denote by $M^{op}(a, i)$ the matrix representing the relation $\{(q, q') \mid q \xrightarrow{(\mathrm{op}, a):\gamma_i} q' \in \Delta\}$, and similarly for $M^{cl}(a, i)$. Finally, for each $j \leqslant |Q|$ and each array v of $|Q|$ booleans, we denote by $C(j, v)$ the square matrix whose j^{th} column is v, with other columns being zero (i.e., with all elements equal to false).

We next describe a first strategy to compute the relations Update^a_S. For $i \leqslant |\Gamma|$, $j \leqslant |Q|$, and each array v of $|Q|$ booleans, we compute the product $R(i, j, v)$ of the three boolean square matrices $M^{op}(a, i)$, $C(j, v)$ and $M^{cl}(a, i)$, in time $O(|Q|^2)$. We then compute the sum $R'(j, v)$ of all matrices $R(i, j, v)$, in $O(|\Gamma| \times |Q|^2)$. As this is done for all j and v, the total time for that preprocessing is $O(|\Sigma| \times |Q|^3 \times |\Gamma| \times 2^{|Q|})$. With this datastructure, we can compute Update^a_S in $O(|Q|^3)$ for any set $S \subseteq Q^2$ and $a \in \Sigma$: we simply sum all matrices $R'(j, v_j)$ with $j \leqslant |Q|$, and v_j the j^{th} column of S.

We could also adopt another strategy to compute the relations Update^a_S. For every column array v of $|Q|$ booleans, every $a \in \Sigma$ and every $i \leqslant |\Gamma|$, we compute the product of $M^{op}(a, i)$ with v. For every row array v of $|Q|$ booleans, every $a \in \Sigma$ and every $i \leqslant |\Gamma|$, we compute the product of v with $M^{cl}(a, i)$. The total time for this preprocessing is $O(|\Sigma| \times |\Gamma| \times |Q|^3 \times 2^{|Q|})$. With this datastructure, we can compute Update^a_S in $O(|Q|^2 \times |\Gamma|)$ for any set $S \subseteq Q^2$ and $a \in \Sigma$: we compute for each $i \leqslant |\Gamma|$ the product $R''(i, S)$ of the three matrices $M^{op}(a, i)$, S, and $M^{cl}(a, i)$. For each i, this can be achieved in $O(|Q|^2)$ using the datastructure from preprocessing. Then one sums all matrices $R''(i, S)$ for all $i \leqslant |\Gamma|$, in time $O(|Q|^2 \times |\Gamma|)$. We have thus given two methods that allow to process each transition in $O(|Q|^3)$, or $O(|Q|^2 \times |\Gamma|)$ after a preprocessing in total time $O(|Q| \times Q^{2|Q|} + |\Sigma| \times |\Gamma| \times |Q|^3 \times 2^{|Q|})$. But the $|\Sigma| \times |\Gamma|$ factor only takes into account pairs that appear in some transitions, so it can be bounded by $|\mathcal{A}|$. We can therefore use $O(|\mathcal{A}| \times 2^{2|Q|})$ as an upper bound for the preprocessing. Let us now briefly survey some related results from the literature on the evaluation of VPAs before we introduce the problem of emptiness in presence of a DTD.

In terms of complexity classes, emptiness of VPAs is PTime-hard [Lan11], and Alur and Madhusudan [AM09] provide several bounds showing that membership can be solved with small space and time requirements (typically sublinear space) for a fixed automaton. But we only focus on the degree of the polynomial involved and do not consider other indicators for the complexity such as complexity classes or circuit complexity. It should also be noted that the “trick” of using fast-matrix multiplication has been used early on in the related problem of parsing context-free grammars: Valiant observed that the membership problem for context-free grammars can be solved in subcubic time using matrix-multiplication [Val75].

As already mentioned, Alur and Madhusudan [AM09] evaluate to $O(|\mathcal{A}|^3)$ and $O(|\mathcal{A}|^3 \times |t|)$ the complexity of the emptiness and evaluation problems for a VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ and document t. The construction in [Gau09] as discussed above allows to refine those complexities to $O(|\Delta|^2 + |Q|^3)$ and $O((|\Delta|^2 + |Q|^3) \times |t|)$. La Torre et al. [TNP07] investigate the complexity of membership for visibly pushdown languages described by so-called *visibly pushdown grammars*, and give algorithms deciding $w \in L(G)$ in $O(|G| \times |w|)$ for every such grammar G and nested word w. The transformation of VPAs into visibly pushdown grammars is polynomial but excessively expensive. We also detailed an algorithm to check emptiness of VPAs using the configuration automata ($\mathcal{P}$-automata), along the lines of [Tan09]. The OpenNWA implementation of VPAs by Driscoll et al. [DTR] also relies on the configuration automata for testing emptiness, which is quite natural since the library is derived from a more general project on pushdown systems.

We are not aware of better bounds in the literature. Certainly, a bound in $O((|Q| + |\Gamma|)^3 \times |w|)$ has been suggested for the emptiness problem in [AM04b], and a bound in $O(|Q|^2 \times |\Gamma| \times |w|)$ has been mentioned for the evaluation problem in [TNP07], but from private communications with the authors, it seems these bounds cannot be sustained, although we have no argument against their correctness. Another estimation of the complexity can be found in [TVY08], but it is at best incomplete: complexity $O(|Q|^2 \times |\Sigma| \times |w|)$ is claimed, but not supported by further justification[a].

There have been several implementations of VPAs, as evidenced on the web page by Madhusudan dedicated to visibly pushdown automata literature [vpa]. Several of these implementations provide general constructions (intersection...) on VPAs, and allow to decide classical problems such as evaluation, inclusion... We mention in particular three libraries (not yet mentioned on the VPA web page) that emphasize performance for typical applications of VPAs: XEvolve [PSZ11] shows that VPAs provide space efficient solutions for the validation and typechecking of XML schemata, whereas FXP [DGN+12] and XSeq [MZZ12]. focus on the efficient evaluation of XPath languages.

[a] In particular, it is surprising that neither the number of stack symbols nor the number of transitions appear in the formula: testing in constant time the emptiness of a VPA with, say, three states over a one-letter alphabet, seems unfeasible if the number of transitions and stack symbols is arbitrary.

Emptiness for VPAs in Presence of a DTD We extend the algorithm deciding emptiness to support satisfiability under a schema constraint expressed with a DTD, where DTDs are defined on page 84. The problem of deciding emptiness under a DTD constraint and the problem of VPA evaluation are related in the sense that given any tree t and VPA $\mathcal{A}$, we can build in $O(|t| + |\mathcal{A}| \times |t|)$ a DTD D of size $O(|t|)$ and VPA $\mathcal{A}'$ of size $O(|\mathcal{A}| \times |t|)$ having the same states than $\mathcal{A}$, such that $L(\mathcal{A}') \cap L(D) = \varnothing$ if and only if $t \in L(\mathcal{A})$: we simply use alphabet N_t and replace each transition of $\mathcal{A}$ using letter a by one transition for each node of t labeled with a.

Of course, one can decide if the languages of a VPA and a DTD have empty intersection by first translating the DTD into an equivalent VPA, computing a VPA for the intersection, and then testing emptiness of the resulting automaton, but we can obtain better complexity.

Proposition 3.10. *Given a VPA $\mathcal{A} = (\Sigma, Q_\mathcal{A}, \Gamma_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \Delta_\mathcal{A})$ and DTD D, one can decide if there exists some tree $t \in L(D) \cap L(\mathcal{A})$ in time $O(|\Delta_\mathcal{A}| \times |Q_\mathcal{A}| + |Q_\mathcal{A}|^3 \times |D| \times |\Sigma| + |Q_\mathcal{A}|^2 \times |D|^2 \times |\Sigma|)$.*

Proof. Let $\mathcal{A} = (\Sigma, Q_\mathcal{A}, \Gamma_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \Delta_\mathcal{A})$ a VPA and $D = (r, \Sigma, P)$ a DTD. We begin with the construction of one Glushkov automaton for each letter $a \in \Sigma$, and denote by i_a the initial state of the automaton corresponding to the production from letter a (a unique state indeed, according to the definition of Glushkov automata). We use distinct states for each automaton, and denote by Q_D the union of the states, by I_D the unions $\bigcup_{a\in\Sigma}\{i_a\}$ of all initial states and by Δ_D the union of the transition rules for all these automata. Q_D and Δ_D can clearly be computed in time $O(|D^2| \times |\Sigma|)$, and even in $O(|D| \times |\Sigma|)$ for XML DTDs.

The simplest approach to test satisfiability of $\mathcal{A}$ with respect to DTD D may be to build a VPA for the intersection of $L(\mathcal{A})$ and D by the standard product construction. This however results in a VPA with state set $Q_\mathcal{A} \times Q_D$. The complexity obtained from Proposition 3.8 in that case would still be polynomial but raises the degree of the polynomial to 6 as it includes the term $(Q_\mathcal{A} \times Q_D)^3$. A more careful analysis of the reachability relation allows to lower the complexity: we define a relation $R \subseteq Q_\mathcal{A}^2 \times Q_D$ such that for any $q_1, q_2 \in Q_\mathcal{A}$ and $q \in Q_D$, (q_1, q_2, q') belongs to R if and only if there exist $n \geqslant 0$ trees $t_1, \ldots, t_n$ with root symbols $a_1, \ldots, a_n$, satisfying the following three conditions:

- $\mathcal{A}$ admits a run from q_1 to q_2 over the hedge $t_1 \ldots t_n$,
- there exist $q'_0, q'_1, \ldots, q'_n = q'$ such that for every $j < n$, $(q'_j, a_{j+1}, q'_{j+1})$ belongs to Δ_D and $q'_0 \in I_D$
- for every $j \leqslant n$, t_j satisfies the DTD (a_j, Σ, P).

The relation R can be computed according according to the following rules:

1. $(q, q, i_a) \in R$ for all $a \in \Sigma$
2. $(q_0, q_4, q'') \in R$ for all $q_0, q_1, q_2, q_3, q_4 \in Q_\mathcal{A}$, $q, q', q'' \in Q_D$, $\gamma \in \Gamma_\mathcal{A}$, and $a \in \Sigma$ satisfying the following 6 conditions: (1) $(q_0, q_1, q) \in R$, (2) $q_1 \xrightarrow{(\mathrm{op},a):\gamma} q_2 \in \Delta_\mathcal{A}$, (3) $q_3 \xrightarrow{(\mathrm{cl},a):\gamma} q_4 \in \Delta_\mathcal{A}$, (4) $(q_2, q_3, q') \in R$, (5) $q' \in F_a$, (6) $(q, a, q'') \in \Delta_D$.

The 6 conditions of the saturation rule 2 can be decomposed as follows into auxiliary relations R_0 and R_1 to obtain the complexities stated above: $R_0(q_1, q_4, a)$ if (2),(3),(4) and (5) are satisfied $R_1(q_0, q_4, a, q)$ if (1) is satisfied and $R_0(q_1, q_4, a)$, and $R(q_0, q_4, q'')$ if (6) is satisfied and $R_1(q_0, q_4, a, q)$. To conclude our proof, we observe that $L(D) \cap L(\mathcal{A}) \neq \varnothing$ iff there exists (q_1, q_2, q') in R, $q_i \in I_\mathcal{A}$, $q_f \in F_\mathcal{A}$, and $\gamma \in \Gamma$ such that $q_i \xrightarrow{(\mathrm{op},r):\gamma} q_1 \in \Delta$, $q_2 \xrightarrow{(\mathrm{cl},r):\gamma} q_f \in \Delta$, and $q' \in F_r$. □

Remark 3.3. *For XML DTDs, the complexity can be lowered to* $O(|\Delta_\mathcal{A}| \times |Q_\mathcal{A}| + |Q_\mathcal{A}|^3 \times |D| \times |\Sigma| + |Q_\mathcal{A}|^2 \times |D| \times |\Sigma|)$ *since there exists at most one* q'' *such that* $(q, a, q'') \in \Delta_D$ *for each pair* (q, a) *in* $Q_D \times \Sigma$.

3.2.3. Pumping Lemmas for VPAs

We will use pumping arguments in several proofs. We essentially distinguish two different pumping arguments on unranked trees: one can either replace the subtree at a node with the subtree below a descendant of this node or delete a sequence of consecutive subtrees rooted at a same node n. In the first case we obtain a tree of strictly smaller depth, and in the second case, we lower the number of children of node n. Let us detail those two transformations.

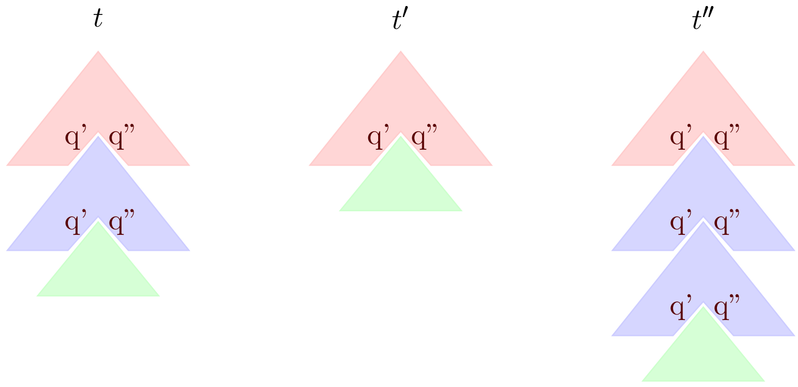


Figure 3.7.: Vertical pumping lemma for VPAs.

Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA, t a tree in $L(\mathcal{A})$, ρ an accepting run of $\mathcal{A}$ in t and n a node in t, the pair of states $\rho^{\uparrow}(n) =$ (q’, q”) characterizes the subtrees (or hedges) that could be used to replace the subtree rooted at n in t without modifying the run in every other node of t. The tree obtained from t by replacing node n and all its descendants with hedge h belongs to $L(\mathcal{A})$ for all h in Ł$(\mathcal{A}_{q_1,q_2})$. Consequently we obtain the following lemma:

Lemma 3.11. *Let $\mathcal{A}$ a VPA, t a tree in $L(\mathcal{A})$ and ρ an accepting run of $\mathcal{A}$ on t. If there are nodes $n \neq n'$ in t with $n \preccurlyeq_t n'$ and $\rho^{\uparrow}(n) = \rho^{\uparrow}(n')$, then the trees t' and t'' also belong to $L(\mathcal{A})$, where t' is the tree obtained from t by replacing the subtree rooted at n (n included) by the subtree rooted at n' whereas t'' is obtained from t by repeating the "part" of t between n and n'.*

Remark 3.4. *The two trees t' and t'' are defined rather informally in the lemma above. Set $t = (\Sigma_t, N_t, child_t, follow_t, lab_t)$, with $n, n' \in N_t$ such that $n \preccurlyeq_t n'$. The tree t' is then defined by $t' = (\Sigma_t, N_{t'}, child_{t'}, follow_{t'}, lab_{t'})$ where $N_{t'} = \{x \in N_t \mid n' \preccurlyeq_t x \vee n \not\preccurlyeq_t x\}$, $lab_{t'}$ is the restriction of lab_t to the nodes in t', $child_{t'} = (child_t \cap N_{t'}^2) \cup \{(Parent_t(n), n')\}$ and $follow_{t'} = (follow_t \cap N_{t'}^2) \cup \{(y, n') \mid (y, n) \in follow_t\} \cup \{(n', y) \mid (y, n) \in follow_t\}$. The tree t'' is defined in a similar way.*

Let Q denote the states of the VPA. As soon as $depth(t) > |Q|^2$, tree t has two nodes n, n' on which this pumping lemma can be applied. Therefore, the following result is an immediate corollary of Lemma 3.11. Observe that this result is essentially optimal as the proof of Proposition 3.7 provides a VPA of size $O(n)$ that accepts no tree of depth less than $n(n-1)$.

Proposition 3.12. *Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA, if $L(\mathcal{A})$ is not empty then every tree of minimal size in $L(\mathcal{A})$ has depth at most $|Q|^2$.*

Let us now focus on the horizontal pumping argument. It essentially states that in any accepting run of $\mathcal{A}$ over a nested word of minimal size from

$L(\mathcal{A})$, $\mathcal{A}$ cannot reach twice the same configuration. Let $\mathcal{A}$ denote a VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$, t a tree in $L(\mathcal{A})$ with $lin(t) = a_1a_2 \dots a_m$, n a node in t and $\rho = (q_0, \sigma_0) \dots (q_m, \sigma_m)$ an accepting run of $\mathcal{A}$ over t. Let a_i and a_j the pair of opening and closing tags corresponding to n. For every $k \leqslant m$, the pair (q_k, σ_k) with the current state and stack contains all the "relevant information" from ρ. Therefore, for every tree t' with $lin(t') = b_1 \dots b_{m'}$ and every accepting run $(q'_0, \sigma'_0) \dots (q'_{m'}, \sigma'_{m'})$ of $\mathcal{A}$ over t', if there exists some natural $k' \leqslant m'$ that satisfies $(q_{k'}, \sigma_{k'}) = (q_k, \sigma_k)$, then the nested word $a_1 \dots a_{k-1}b_{k'} \dots b_{m'}$ belongs to $L(\mathcal{A})$ under the enforced tree-input hypothesis. As a corollary, we obtain Proposition 3.13.

Proposition 3.13. *Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA, if $L(\mathcal{A})$ is not empty then every tree of minimal size in $L(\mathcal{A})$ contains no node with $|Q|$ children or more.*

The pumping argument above also shows that in any tree of minimal size from $L(\mathcal{A})$ the number of nodes at depth $k \geqslant 1$ is at most $1 + \min((s-1)|\Gamma|^{k-1}, (s-1)^k)$, where s denotes the number of states $|Q|$.

Corollary 3.14. *Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA. If $\mathcal{A}$ accepts a tree of depth k, then it accepts some tree of size at most $1 + (|Q| - 1)^k$.*

We observe that both Propositions 3.12 and 3.13 can be interpreted in terms of the usual pumping lemma for ranked trees using fcns encoding: for Proposition 3.12, the pumping argument deals with two nodes one of which is below the left child of the other in the fcns encoding, and for Proposition 3.12, the pumping argument deals with two nodes one of which is below the right child of the other in the fcns encoding. Combining Propositions 3.12 and 3.13, we obtain a rough bound on the size of the smallest tree accepted by $\mathcal{A}$ when $L(\mathcal{A})$ is not empty, but we can obtain a finer pumping lemma by combining horizontal and vertical pumping into a single pumping argument. This combination is essentially obtained from the vertical pumping lemma, using pumping arguments over hedges instead of trees. Let t a tree and $n_{\leftarrow}, n_{\rightarrow}, n_{\swarrow}, n_{\searrow}$ four nodes of t, not necessarily distinct, satisfying the following four conditions: (1) $n_{\rightarrow}$ is a following sibling of $n_{\leftarrow}$, (2) $n_{\searrow}$ is a following sibling of $n_{\swarrow}$, (3) $n_{\leftarrow}$ precedes $n_{\swarrow}$ in document order, and (4) $n_{\searrow}$ precedes $n_{\rightarrow}$ in document order. Let $a_1 \dots a_m$ the linearization of t, and let $i_1 \leqslant i_2 < i_3 \leqslant i_4$ denote the positions of respectively the opening tags of $n_{\leftarrow}$ and $n_{\swarrow}$, and the closing tags of $n_{\searrow}$ and $n_{\rightarrow}$. Suppose additionally that $\mathcal{A}$ has an accepting run over t of the form $(q_0, \sigma_0) \dots (q_m, \sigma_m)$ such that both $q_{i_1} = q_{i_2}$ and $q_{i_3} = q_{i_4}$. Then $a_1 \dots a_{i_1-1}a_{i_2} \dots a_{i_3}a_{i_4+1} \dots a_m$ belongs to $L(\mathcal{A})$.

As a corollary of this result we easily obtain Proposition 3.15, which is also the bound obtained when combining the standard pumping lemma over ranked trees with the transformation from VPAs to NTAs via fcns encoding discussed in Proposition 3.7. Nevertheless the pumping argument over the fcns encoding yields a particular case of our pumping argument, obtained by restricting $n_{\searrow}$ and $n_{\rightarrow}$ to be the rightmost children of their parent.

Proposition 3.15. *Let $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ a VPA. If $L(\mathcal{A})$ is not empty then $\mathcal{A}$ accepts a tree of size at most $2^{|Q|^2}$.*

This upper bound is essentially optimal, as we show in the next proposition:

Proposition 3.16. *For every n, there exists a VPA $\mathcal{A}_n$ with n states such that the smallest tree in $L(\mathcal{A}_n)$ has size $2^{\Omega(n^2)}$.*

Proof. See the appendix, page 275. □

3.2.4. Schema Languages for XML

DTDs and EDTDs A *Document Type Definition* (DTD) is a set of declarations that define the type of the document. Formally, we model a DTD as a triple $D = (\Sigma, r, P)$ where Σ is the alphabet, $r \in \Sigma$ is the root symbol, and P is the set of rules, i.e., a function that maps Σ to regular expressions over Σ. In the sequel, we write DTD rules as $a \to e$ and if for a symbol a the rule is not specified, then $a \to \varepsilon$ is implicitly assumed.

The dependency graph of a DTD $D = (\Sigma, r, P)$ is a directed graph whose node set is Σ and whose set of edges contains (a, b) iff $P(a)$ uses the symbol b. A DTD is *recursive* iff its dependency graph is cyclic. The size $|D|$ of a DTD $D = (\Sigma, r, P)$ is the sum of the sizes of the regular expressions $P(\alpha)$ appearing in D. A tree t *satisfies* a DTD $D = (\Sigma, r, P)$ if its root is labeled r and, for every node n with k children $n_1, \ldots, n_k$ (listed in the document order), we have $lab_t(n_1) \cdots lab_t(n_k) \in L(e)$, where $e = P(lab_t(n))$. By $L(D)$ we denote the set of all trees that satisfy D. In terms of expressive power, DTDs cannot express all regular tree languages. A severe limitation is the absence of a subtyping mechanism: in general we cannot define with a DTD the union of two DTDs [PV00]. From a language-theoretic point of view, the languages defined by DTDs correspond to the class of *local* regular tree languages [MLMK05]. In this dissertation, we extend the definition of local tree languages to non-regular languages: we say a language L is *local* if it satisfies the following subtree exchange property: for any pair of trees $t, t' \in L$ and for any nodes n in t and n' in t', if n and n' share the same label, then L also contains the tree obtained from t by replacing the subtree below n with the subtree below n' from t'. This means that the subtrees allowed below a node only depend on the label of the node: if $L(D)$ contains two trees t and t' with a-labeled nodes $n \in N_t$ and $n' \in N_{t'}$, then t remains in $L(D)$ if we replace its subtree below n ($t_{\restriction n}$) by $t'_{\restriction n'}$.

Extended DTDs, were proposed by Papakonstantinou and Vianu [PV00] under the denomination of *specialized DTD*, in order to overcome the limitations of DTDs in terms of expressive power. They enhance DTDs with a subtyping mechanism, which allows to define any regular tree language. Formally, an *extended DTD* is a tuple $\mathcal{E} = (\Sigma, \Sigma', D, \mu)$ with Σ, Σ' two alphabets, D a DTD over Σ', and μ a function from Σ' to Σ. A tree t belongs to $L(\mathcal{E})$ if and only if there exists some tree t' over Σ' such that $t' \in L(D)$ and $\mu(t') = t$, where $\mu(t')$ is the tree obtained by relabeling every a-labeled node n of t' with $\mu(a)$, for every $a \in \Sigma'$.

Murata et al. [MLMK05] defined several subclasses of EDTDs deserving attention for the modelization of XML schema languages. In particular, an EDTD $\mathcal{E} = (\Sigma, \Sigma', D, \mu)$ is *single-type* if for every $a \in \Sigma'$ and for every pair of symbols $x, y \in \Sigma'$ that occur in $D(a)$, $\mu(x) = \mu(y)$ implies $x = y$. An EDTD is *restrained competition* if for every $a \in \Sigma'$ and for every pair of symbols $x, y \in \Sigma'$ and every words wxu and wyv in $D(a)$, $\mu(x) = \mu(y)$ implies $x = y$.

In Section 6.3, we will also use two unorthodox kinds of schemata: a *Context-free DTD* (CDTD) is a DTD in which the production rules can use context-free grammars instead of regular expressions, whereas an *Extended Context-free DTD* (ECDTD) adds a subtyping mechanism to Context-free DTDs[6]. Formally, a Context-free DTD is a triple $D_0 = (\Sigma, r \in \Sigma, P)$ where P, the set of rules, maps Σ to a CFG over Σ. An Extended Context-free DTD is a tuple $\mathcal{E} = (\Sigma, \Sigma', D, \mu)$ with D a Context-free DTD over Σ', and μ a function from Σ' to Σ. The definition for the languages of D_0 and $\mathcal{E}$ is similar to the corresponding definition for DTDs and EDTDs.

SGML/XML DTDs In SGML/XML terminology, the production $P(\alpha)$ of a symbol α is usually called the *content model* of α. For compatibility with SGML, XML DTDs are required to use deterministic content model. Formally, this requirement means that regular expression $P(\alpha)$ has to be deterministic for every symbol α [XML99]. We will call *XML DTD* a DTD satisfying this determinism constraint, and keep the "DTD" denomination for our more permissive schema definition disregarding this constraint. This denomination is slightly abusive since our model for XML DTDs leaves out several feature of real DTDs.

XML Schema XML schema definition language (or XML Schema, in short) is the XML-based schema language proposed by the W3C [XML04]. It enhances DTDs with a typing mechanism and allows to express richer constraints on the content of elements. We should mention several other features from XML Schema which we do not consider. For instance, dependencies are supported by XML Schema: elements `unique`, `key` and `keyref` allow to express integrity constraints similar to the unique, primary and foreign key

[6] In the terminology of [PV00], these would be called "context-free ltds" and "specialized context-free ltds". These should not be confused with extended context free grammars, which are context-free grammars in which the right-hand side of the productions may use regular expressions instead of a single word.

constraints in the relational model. Also some (restricted) form of unordered concatenation can be used to specify the elements appearing below another element, through the `all` groups. Furthermore, it supports namespaces and uses an XML syntax, unlike DTDs. The number of occurrences of an element or group of elements below another element can be specified in XML Schema through attributes `minOccurs` and `maxOccurs`.

XML Schema imposes a constraint named the “Element Declarations Consistent” which Murata et al. formalize via the single-type restriction on EDTDs [MLMK05]. The efficiency of validation provides a rationale for this constraint: for a single-type EDTD $\mathcal{E}$, types can be attributed to the nodes in a top-down traversal. Thus, a top down algorithm allows to test if a tree t belongs to $L(\mathcal{E})$.

To further facilitate the validation, XML Schema imposes a constraint similar to the determinism of regular expressions: the “Unique Particle Attribution”. This constraint is more tricky to verify than determinism of regular expressions due to the richer structure of XML Schema content models: expressions with numeric occurrences require special care. It remains easier to check than its SGML counterpart, however, because interleaving is very restricted in XML Schema.

Martens et al. [MNSB06] argue that these two constraints in XML Schema could be relaxed while preserving the efficiency of the validation/typing algorithms. They propose a more liberal constraint characterizing the EDTDs that allow to attribute the type of an node at his opening tag (in a streaming traversal of the document). They characterize the EDTDs satisfying this property, and christen them “*one pass preorder-typeable* EDTDs”. The authors show that EDTDs admit one pass preorder typing if and only if its trimmed version is restrained-competition. They also provide semantic characterizations for single-type and restrained-competition EDTDs, and study the complexity of the following three classes of problems: (1) deciding if an EDTD is single-type or one pass preorder typeable, (2) deciding if an EDTD can be simplified into an equivalent EDTD in those classes (3) deciding the containment problem for EDTDs in those classes. In particular for (2), the authors prove by reduction to the universality problem for NTAs that deciding if an EDTD admits an equivalent DTD is EXPTIME-complete, and similarly for simplification into equivalent one pass preorder-typable or restrained-competition EDTDs.

We investigate in Section 6.3 the validation of the document against a schema, when the schema is modeled as a DTD with deterministic regular expressions, but except for determinism issues in Chapter 6, we will only consider schemata defined by general tree automata or DTDs.

Alternative Schema Languages for XML In spite of its powerful typing and its good integration within the XML languages, the XML Schema Definition language has not superseded the older but simpler DTDs. XML Schema has been criticized among other for its complexity [Cla02]. A prominent alternative to XML Schema is Relax NG [Rel01] from the OASIS consortium and based on languages by J.Clark and M.Murata. It shares many features with XML Schema. In particular, it supports namespaces and also provides a typing mechanism. However, it is closer to regular tree language formalisms such as EDTDs, and admits both an xml syntax and a more compact non-xml syntax. Furthermore it does not require deterministic content models, and shows better support for unordered content models. Those construction, however, raise the complexity of the validation.

Schematron is a rule-based schema language and is specified as an ISO/IEC standard. Each rule is an XPath expression expressing a constraint that must be satisfied by the document. Thus, relations between distant parts of the document (patterns) can easily be described in this language. However, structural constraints are often best described by grammar-based formalisms, so that this schema language finds a natural use in conjunction with another schema such as Relax NG or XML Schema.

DataGuides provide another way to type graph data to facilitate the formulation of queries and their optimization [GW97], in particular when a schema is not available. DataGuides were introduced for semistructured data based on the OEM model and have since also been used in XML context. A DataGuide is an automaton (DFA) representation of all the paths from the root of the document. For a given finite graph, this set is regular. One of the issues raised by DataGuides and investigated by Goldman and Widom is the efficient computation and maintenance of DataGuides.

3.3. Query Languages, Views and Updates

Queries can be defined independently of the way they are specified. Modulo technical details, a query will be any function that takes as input a tree and returns a set of selected nodes.

Definition 3.2. *A* (unary) query *is a function Q that maps a document t to a set of nodes $Q(t) \subseteq N_t$.*

We slightly amend this definition as we only consider queries (and a fortiori views) closed under isomorphism. Assuming queries to be closed under isomorphism is justified whenever we use automata or XPath as query languages, but it can be rather limiting if we wish to match (test equality) of a node id against a constant, which could be expressed with minor adaptations of our formalisms. Therefore, we will explicitly specify which results assume queries to be closed under isomorphism.

The *domain* $\mathrm{dom}(Q)$ is the set of trees t in T_Σ such that $Q(t)$ is not empty. A query Q is *root-preserving* if for every $t \in T_\Sigma$, either $Q(t) = \varnothing$ or

$root_t \in Q(t)$. A *Boolean query* returns a Boolean value instead of a set of nodes: $Q(t) \in \{\mathsf{true}, \mathsf{false}\}$. We sometimes write $t \models Q$ for $Q(t) = \mathsf{true}$, and $t \not\models Q$ for $Q(t) = \mathsf{false}$.

Containment, equivalence and satisfiability are classical decision problems for query languages. They are defined as follows:

Problem: Containment: $Q_1 \subseteq Q_2$

Input: two (unary) queries Q_1 and Q_2
Question: *Does it hold that for every tree t, $Q_1(t) \subseteq Q_2(t)$?*

Problem: Equivalence: $Q_1 \equiv Q_2$

Input: two (unary) queries Q_1 and Q_2
Question: *Does it hold that for every tree t, $Q_1(t) = Q_2(t)$?*

Problem: Satisfiability

Input: a Boolean query Q
Question: *Is there any tree t such that $Q(t) = \mathsf{true}$?*

Containment and Equivalence are usually "hard" decision problems for expressive query languages. Hardness results for those problems will provide us with a few lower bounds on policy comparison. The *model checking* problem is the problem of evaluating a Boolean formula on the document. It therefore provides a lower bound for the complexity of evaluating unary queries. Formally, the model checking problem takes as input a tree t and Boolean formula ϕ, and decides if $t \models \phi$. We will also study the problem of satisfiability under non-recursive DTDs, that takes as input a Boolean query Q and a non-recursive DTD D, and decides if there exists a tree $t \in L(D)$ such that $Q(t) = \mathsf{true}$.

3.3.1. First Order and Monadic Second Order Logic

In this dissertation we scarcely use logical formalisms apart from XPath-based languages. Nonetheless, it seems relevant to present those formalisms as they lie at the core of both automata and query languages.

Syntax and Semantics: FO and MSO A First-order logic (FO) formula over signature $\sigma = (child, follow)$ is a logical formula defined by the following grammar:

$$\phi \coloneqq lab(x) = a \mid \exists x.\phi \mid \forall x.\phi \mid child(x,y) \mid follow(x,y) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi$$

Monadic Second Order logic (MSO) extends First Order logic with second-order variables, i.e., quantification over sets. The grammar defining second-order logic formulae is obtained from the grammar above through the addition of the following rules:

$$\phi \coloneqq \exists X.\phi \mid \forall X.\phi \mid x \in X$$

These are actually the definition of *FO* and *MSO* over unranked trees seen a relational structure with relations *child* and *follow*. First and Second Order formulae over words can be defined similarly, removing the *child* relation from the signature.

The semantics of a formula is defined according to an interpretation of its free variables, i.e., a mapping from the free variables to nodes or sets of nodes. We fix a tree t, naturals $k, k' \in \mathbb{N}$, a *MSO* formula with k free first-order variables $x_1, \ldots x_k$, and k' free second-order variables $X_1, \ldots X_{k'}$. Let I a mapping such that for every $i \leqslant k, I(x_i) \in N_t$ and for every $j \leqslant k'$, $I(X_j) \subseteq N_t$. Then the notation $t, I \models \phi$ denotes the judgment: "formula ϕ is satisfied on tree t under interpretation I". The semantics of formulae is defined as in Figure 3.8:

$$\begin{array}{lll}
t, I \models \mathit{lab}(x) = a & \iff & \mathit{lab}_t(I(x)) = a, \\
t, I \models x \in X & \iff & I(x) \in I(X), \\
t, I \models \mathit{child}(x, y) & \iff & (I(x), I(y)) \in \mathit{child}_t, \\
t, I \models \mathit{follow}(x, y) & \iff & (I(x), I(y)) \in \mathit{follow}_t, \\
t, I \models \phi_1 \vee \phi_2 & \iff & (t, I \models \phi_1) \text{ or } (t, I \models \phi_2) \\
t, I \models \phi_1 \wedge \phi_2 & \iff & (t, I \models \phi_1) \text{ and } (t, I \models \phi_2) \\
t, I \models \neg\phi_1 & \iff & t, I \models \phi_1 \text{ is } \mathsf{false} \\
t, I \models \exists x.\phi_1 & \iff & \text{there is some } n \in N_t \text{ with } t, (I \cup \{x \mapsto n\}) \models \phi_1 \\
t, I \models \forall x.\phi_1 & \iff & \text{for all } n \in N_t\text{: } t, (I \cup \{x \mapsto n\}) \models \phi_1 \\
t, I \models \exists X.\phi_1 & \iff & \text{there is some } S \subseteq N_t \text{ with } t, (I \cup \{X \mapsto S\}) \models \phi_1 \\
t, I \models \forall X.\phi_1 & \iff & \text{for all } n \subseteq N_t\text{: } t, (I \cup \{X \mapsto n\}) \models \phi_1
\end{array}$$

Figure 3.8.: The semantics of *MSO*.

Logical Queries An *MSO* formula ϕ without free variable is a Boolean formula. Then interpretations are useless, and for each tree t, $\phi(t) = \mathsf{true}$ if and only if t is satisfied by the formula, i.e., $t, \varnothing \models \phi$. Thus, formulae without free variables express Boolean queries. Similarly, formula with one free variable express unary queries. Let Q a first order formula with one free variable x, or a monadic second order formula with one free first-order variable x. Then $Q(t) = \{n \in N_t \mid t, \{x \mapsto n\} \models Q\}$.

This logical framework could obviously be extended to allow the definition of binary queries, or even queries of arbitrary arity: a formula with n free (first-order) variables represents a n-ary query. However, this dissertation focuses on unary and Boolean queries so we will not use such queries.

Evaluation of FO and MSO Formula (Model Checking) A major reason why we do not use FO nor MSO queries in our framework is the high complexity involved by reasoning in these logics. The evaluation of a formula is already inefficient: the model checking problem is PSPACE-complete for FO and MSO over trees (and more generally, over finite structures) [Sto74, Var82]. One could resort to the fixed-parameter tractability of MSO and convert the formula into an automaton in order to obtain a complexity linear in t. But the conversion is non-elementary in the size of the formula. What is more, Frick and Grohe [FG04] show under the assumption PTIME ≠ NP that the model checking of MSO on words is not solvable in time $f(|\phi|) \times p(|t|)$ for any elementary function f and polynomial p. Similarly, they show that the model checking of FO on words is not solvable in time $f(|\phi|) \times p(|t|)$ for any elementary function f and polynomial p, unless $\mathrm{FPT} = \mathrm{AW}[*]$.

3.3.2. XPath Dialects

XPath is a language designed by the W3C in order to address parts of an XML document [XPa99]. It is used as a selecting or matching component in several XML query or transformation languages such as XQuery and XSLT, the XPointer framework... In addition to selecting nodes, the evaluation of an XPath query can also return a Boolean value, a string, or a number. Since arithmetic operations in full XPath 1.0 make classical problems such as equivalence or containment undecidable, numerous restrictions have been proposed that yield more tractable fragments of XPath [BK08]. The mainstream approach when studying the usual decision problems for XPath consists in restricting the queries to the navigational core of XPath, leaving out the strings and numbers. We follow this approach and consider only XPath dialects without strings nor arithmetic operations.

NavXPath [BK08] is the basic navigational fragment of XPath, with all four axes: next-child, previous-child, parent, and child, the transitive closure of these axes, path composition and union, and filters. This NavXPath fragment is also referred to as CoreXPath 1.0 [GK02, GKP03] except that usually the next-sibling and preceding-sibling axes are not available in CoreXPath 1.0. Since NavXPath cannot express full First Order Logic on trees of depth 1 (not to mention transitive closure), the language accepted by a DTD cannot even be expressed via an XPath query. Regular XPath [Mar04], which we also denote by $\mathcal{X}Reg$, extends NavXPath with transitive closure, and therefore can express DTD languages. The syntax of Regular XPath is as follows:

$$\begin{aligned}
\alpha &::= \mathsf{self} \mid \Downarrow \mid \Uparrow \mid \Rightarrow \mid \Leftarrow \\
f &::= \mathsf{lab}() = b \mid \chi \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\, f \mid f \;\mathsf{and}\; f \mid f \;\mathsf{or}\; f \\
\mathcal{X} &::= \alpha \mid [f] \mid \mathcal{X}/\mathcal{X} \mid \mathcal{X} \cup \mathcal{X} \mid \mathcal{X}^*
\end{aligned}$$

The semantics of $\mathcal{X}Reg$ is given in Fig. 3.9 except for Boolean connectives which are interpreted in the usual manner. We also use $\alpha{::}b$ as a shorthand for $\alpha/\mathsf{lab}() = b$, and even use a for $\mathsf{self}{::}a$. We also use the symbol $\bigvee$ (resp. $\bigwedge$) to denote sequences of disjunctions (resp. conjunction) indexed by a set: if $S = \{x_1, x_2, x_3\}$ then $\bigvee_{i \in S} f_i$ denotes the expression: f_{x_1} or f_{x_2} or f_{x_3}.

$$\begin{aligned}
[\![\mathsf{self}]\!]_t &= \{(n,n) \mid n \in N_t\}, \\
[\![\Downarrow]\!]_t &= child_t, \\
[\![\Uparrow]\!]_t &= child_t^{-1}, \\
[\![\Rightarrow]\!]_t &= next_t, \\
[\![\Leftarrow]\!]_t &= next_t^{-1},
\end{aligned}
\qquad
\begin{aligned}
[\![\mathcal{X}_1/\mathcal{X}_2]\!]_t &= [\![\mathcal{X}_1]\!]_t \circ [\![\mathcal{X}_2]\!]_t, \\
[\![\mathcal{X}_1 \cup \mathcal{X}_2]\!]_t &= [\![\mathcal{X}_1]\!]_t \cup [\![\mathcal{X}_2]\!]_t, \\
[\![\mathcal{X}^*]\!]_t &= [\![\mathcal{X}]\!]_t^*, \\
[\![[f]]\!]_t &= \{(n,n) \in N_t \mid (t,n) \models f\} \\
(t,n) \models \mathsf{lab}() = a &\text{ iff } lab_t(n) = a, \\
(t,n) \models \mathcal{X} &\text{ iff } \exists n' \in N_t.(n,n') \in [\![\mathcal{X}]\!]_t.
\end{aligned}$$

Figure 3.9.: The semantics of $\mathcal{X}Reg$.

For an expression $\mathcal{X}$ in $\mathcal{X}Reg$, $[\![\mathcal{X}]\!]_t$ is the binary reachability relation on the nodes of t defined by the expression $\mathcal{X}$. By $(t,n) \models f$ we denote that the filter f is *satisfied* at the node n of the tree t. We say that an expression $\mathcal{X}$ is *satisfied* in the tree t if $(t, root_t) \models \mathcal{X}$. Then an expression $\mathcal{X}$ in $\mathcal{X}Reg$ defines a query $Q_{\mathcal{X}}$ where the set of answers to the query $Q_{\mathcal{X}}$ in a tree t is defined as

$$Q_{\mathcal{X}}(t) = \{n \in N_t \mid (root_t, n) \in [\![\mathcal{X}]\!]_t\}.$$

Given an expression $\mathcal{X} \in \mathcal{X}Reg$, we denote by $\mathcal{X}^{-1}$ the expression defined hereunder such that $[\![\mathcal{X}^{-1}]\!] = [\![\mathcal{X}]\!]^{-1}$. Essentially, expression $\mathcal{X}^{-1}$ is obtained from $\mathcal{X}$ by reversing the order of composition as well as all axes at the topmost level, keeping filters unchanged. Formally, $\mathcal{X}^{-1}$ is defined by: $(\mathcal{X}_1/\mathcal{X}_2)^{-1} = \mathcal{X}_2^{-1}/\mathcal{X}_1^{-1}, (\mathcal{X}_1 \cup \mathcal{X}_2)^{-1} = \mathcal{X}_1^{-1} \cup \mathcal{X}_2^{-1}, (\mathcal{X}^*)^{-1} = (\mathcal{X}^{-1})^*$, and $[f]^{-1} = [f]$, whereas $\mathsf{self}^{-1} = \mathsf{self}$, $\Downarrow^{-1} = \Uparrow$, $\Uparrow^{-1} = \Downarrow$, $\Rightarrow^{-1} = \Leftarrow$ and $\Leftarrow^{-1} = \Rightarrow$. We denote by $f_{\mathcal{X}}$ the filter $f_{\mathcal{X}} = \mathcal{X}^{-1}/[\,\mathsf{not} \Uparrow]$. Clearly, a node $n \in N_t$ satisfies filter $f_{\mathcal{X}}$ if and only if $n \in Q_{\mathcal{X}}(t)$.

Example 3.5. *Let* $\mathcal{X} = \Downarrow{::}\texttt{project}/[\Downarrow/\Downarrow{::}\texttt{free}]/\Downarrow^*/[\mathsf{lab}() = \texttt{name}\ \mathsf{or}\ \mathsf{lab}() = \texttt{src}]$. *The nodes selected by this Regular XPath query on tree* t_0 *of Figure 3.1 are* $Q_{\mathcal{X}}(t_0) = \{n_4, n_7, n_{18}, n_{21}\}$ *and from* $\mathcal{X}^{-1}$, *we obtain*[7] *the filter* $f_{\mathcal{X}}$:

$$f_{\mathcal{X}} = [\mathsf{lab}() = \texttt{name}\ \mathsf{or}\ \mathsf{lab}() = \texttt{src}]/\Uparrow^*/[\Downarrow/\Downarrow{::}\texttt{free}]/[\mathsf{lab}() = \texttt{project}]/\Uparrow{::}[\,\mathsf{not} \Uparrow]$$

[7] In order to compute $\mathcal{X}^{-1}$ we must first replace the subexpression $\Downarrow{::}\texttt{project}$ with $\Downarrow/[\mathsf{lab}() = \texttt{project}]$ then apply the transformation as defined above for expression using the non-abbreviated syntax.

Evaluation of Regular XPath Regular XPath formulae can be evaluated in quadratic time, with complexity linear both in the size of the data and the size of the query: given a Regular XPath formula $\mathcal{X}$ and a tree t, $Q_{\mathcal{X}}(t)$ can computed in time $O(|\mathcal{X}| \times |t|)$ [Mar04]. The proof in [Mar04] relies on a result from [AI00] for the evaluation of (Boolean) Propositional Dynamic Logic formulae, and on construction of filter $f_{\mathcal{X}}$. Another approach was proposed in [CGLV09] by first translating in linear time expression $\mathcal{X}$ into an equivalent 2-ATA $\mathcal{A}_{\mathcal{X}}$. The resulting 2-ATA $\mathcal{A}_{\mathcal{X}}$ can then be evaluated over t in time $O(|\mathcal{A}_{\mathcal{X}}| \times |t|) = O(|\mathcal{X}| \times |t|)$.

In this dissertation, we sometimes mention results for fragments of XPath and Regular XPath. Those fragments are defined by the axes and operations they allow. Regular XPath($\Downarrow, \Downarrow^*, \cup, [\ ], \wedge, \neg$) for instance, is the fragment that consists of all Regular XPath expression that may use only downward axes, union of paths, and filters using conjunction and disjunction: upward and horizontal axes are proscribed as well as disjunction, although disjunction is not an issue here since it can be encoded with only linear size increase using conjunction and negation.

XPath 1.0 as Defined in the Standard from the W3C Apart from syntactic differences, our model of XPath differs in several respects from the real XPath 1.0 query language. A first (cosmetic) difference between our query languages and XPath 1.0 is the absence of the `next-` and `preceding-sibling` axes in XPath 1.0. These axes can be simulated in XPath 1.0, however, using the `position()` predicate: the next-sibling axis is equivalent to expression `following-sibling::*[position()=1]`. More relevant is the definition in XPath 1.0 of functions returning integers, such as `position()` and `last()` which allow to manipulate positions of elements, `count` which counts the number of nodes returned by an expression. The standard also supports arithmetic operations on those integers, and defines other functions, such as `id()` that allows to select elements by their identity. The specification of XPath 1.0 by the W3C is rather informal, but Gottlob et al. [GKP05] provide a comprehensive formalization of its semantics.

3.3.3. Expressivity and Decision Problems

Logic and Expressivity The language accepted by a Boolean formula ϕ is $L(\phi) = \{t \mid \phi \models t\}$. A word or tree language L_1 is *MSO definable* if there exists a (Boolean) *MSO* formula ϕ such that $L_1 = L(\phi)$. This definition of definability can be extended to any other class $\mathcal{C}$ of formulae: language L_1 is $\mathcal{C}$ definable if there exists a formula $\phi \in \mathcal{C}$ such that $L_1 = L(\phi)$. Regular tree (resp. word) languages are exactly the *MSO* definable tree (resp. word) languages [TW68]. First order definable word and tree languages form a strict subset of *MSO* definable word and tree languages.

NavXPath is strictly less expressive than *FO* over trees. Marx and de Rijke [MdR05] prove that NavXPath captures exactly the expressivity of First Order formulae using only two variables. Marx also proves [Mar05b, Mar05a] that every expansion of NavXPath that is closed under path complementation can express all *FO* queries. Benedikt and Koch survey those results together with several other results regarding the expressivity of numerous XPath fragments [BK08]. They prove that extending of NavXPath with identifiers, data comparisons and aggregation operations results in a *FO*-complete language. Boolean queries can also be used in order to define the schema and then Regular XPath is powerful enough to express DTDs. It is established in [tCS08, tCS10] that Regular XPath is strictly less expressive than *MSO* over trees: Regular XPath cannot express all regular tree languages. More accurately, the authors introduce an extension of Regular XPath with a subtree relativization operator that has the expressive power of *FO* with monadic transitive closure, and prove that it has the expressive power of nested-tree-walking automata. They prove that nested-tree walking automata cannot accept all regular tree languages, which implies that *FO* with monadic transitive closure is strictly less expressive than *MSO*.

Lemma 3.17 ([Mar04]). *From every DTD D, one can build in linear time a Regular XPath filter f such that for every document t, $t \in L(D)$ if and only if $t \models f$.*

However, Regular XPath is not powerful enough for richer schema languages such as EDTDs since the languages definable with Extended DTDs are exactly the regular tree languages.

Membership of a Language to a Class of Languages Given two classes of languages $\mathcal{C}$ and $\mathcal{C}'$ with $\mathcal{C} \subset \mathcal{C}'$, the problem $Memb(\mathcal{C}', \mathcal{C})$ takes as input a language $L \in \mathcal{C}'$ and returns the truth value of the assertion "$L \in \mathcal{C}$". We know that this problem is undecidable when $\mathcal{C}'$ is the set of context-free grammars and $\mathcal{C}$ is the set of regular word languages. We confine ourselves to instances of $Memb(MSO, \mathcal{C})$, and the *MSO* language can be given indifferently by an automaton or an *MSO* formula because we do not consider complexity, only decidability. Algebraic characterizations have provided characterizations helping to decide $Memb(MSO, \mathcal{C})$ for various restrictions of regular word languages.

A notable example is that of first-order definable word languages. First order languages were proved to define exactly the class of the star-free languages [MP71], i.e., the languages that can be expressed by regular expressions without Kleene star, but using intersection and complementation operators defined by $L(e_1 \cap e_2) = L(e_1) \cap L(e_2)$ and $L(e^c) = \Sigma^* \backslash L(e)$. Schützenberger also characterized star-free definable word languages as the regular languages whose syntactic monoid is aperiodic [Sch65]. Therefore, it can be decided if a regular language is first order definable (and the problem PSPACE-complete from a DFA representation [CH91]). Similarly, decidable characterizations have been obtained for numerous classes of word languages. For instance, Therien and Wilke provide a decidable characterization of First Order formulae with two variables (over words) [TW98]. They actually give two characterizations, because when only two variables are allowed, whether the "successor" (next-sibling) predicate is available or not in addition to its transitive closure makes a difference.

This kind of algebraic characterization seems harder to establish for trees, because of the multiple kinds of tree models (ranked, unranked), because of the multiplicity of axis predicates which greatly increase the number combinations that have to be considered, and because of the lack of standard formalism, as argued in [Pla10]. This may explain why few results are known for the problem $Memb(MSO, \mathcal{C})$ when $\mathcal{C}$ is the class of languages definable in expressive XPath dialects, with a few exceptions. Let us mention for instance [BS09] which gives a polynomial algorithm to decide if the language of a bottom-up deterministic automaton is definable in First-order logic (for ranked trees and unordered unranked trees), and [PS10], which gives a decidable characterization for First Order formulae with two variables using ancestor and following-sibling axis. To the best of our knowledge, it is still an open question whether $Memb(MSO, \mathcal{C})$ is decidable when $\mathcal{C}$ is the set of Regular XPath definable tree languages.

Decision Problems for Logical Queries Given a unary query Q, we denote by $Filt(Q)$ the query (or filter) such that for every tree t and node $n \in N_t$, $n \in Q(t)$ if and only if $t, n \models Filt(Q)$. Clearly, for a formula Q expressed in FO, MSO or Regular XPath, we can in linear time compute a formula $Filt(Q)$ in the same language.

Remark 3.5. *For query languages such as FO, MSO or Regular XPath, equivalence, satisfiability and containment are inter-reducible, using negation, intersection and union operations. For instance,* $Q_1 \subseteq Q_2$ *if and only if* $Q_1 \cup Q_2 \equiv Q_2$*, while* $n \in Q_1(t) \backslash Q_2(t)$ *implies* $t, n \models (Filt(Q_1) \wedge \neg Filt(Q_2))$*.*

For NavXPath, the problem of satisfiability is EXPTIME-complete. Satisfiability for NavXPath remains EXPTIME-complete in presence of a schema given by a DTD, and becomes PSPACE-complete if the schema is given by a non-recursive DTD [BFG08]. Satisfiability is EXPTIME-complete for Regular XPath, and this complexity still holds in presence of a DTD as every DTD can be represented with an equivalent Regular XPath formula of linear size [Mar04]. Over a non-recursive DTD, however, or more generally when the depth of the trees (satisfying the formula) is polynomially bounded in the size of the formula, the lower bound does not hold. We

	NavXPath	$\mathcal{X}Reg$	2-ATA	VPA	*FO*, *MSO*
Evaluation	$O(\lvert Q\rvert \times \lvert t\rvert)$	$O(\lvert Q\rvert \times \lvert t\rvert)$	$O(\lvert \mathcal{A}\rvert \times \lvert t\rvert)$	$O(\lvert \mathcal{A}\rvert^3 \times \lvert t\rvert)$	PSPACE-c
Satisfiability	EXPTIME-c	EXPTIME-c	EXPTIME-c	$O(\lvert \mathcal{A}\rvert^3)$	non-elem
Satisf. over non-rec DTD	PSPACE-c	PSPACE-c	PSPACE-c	PTIME	non-elem

Figure 3.10.: Complexity of satisfiability and evaluation

prove in Section 4.2 that satisfiability becomes PSPACE-complete in this setting. Our proof even gives PSPACE-completeness for emptiness of 2-ATAs in this setting. The other bounds for ATAs and VPAs are surveyed in Section 3.2. In particular Propositions 3.9 and 3.8 give refined bounds for the complexity of evaluation and satisfiability, whereas Proposition 3.10 states that for any VPA $\mathcal{A}$ and DTD D over alphabet Σ, one can check if there is a document that belongs simultaneously to $L(\mathcal{A})$ and D with complexity $O(|\Delta_{\mathcal{A}}|^2 + |Q_{\mathcal{A}}|^3 \times |D| \times |\Sigma| + |Q_{\mathcal{A}}|^2 \times |D|^2 \times |\Sigma|)$, for non-recursive (and therefore also for non-recursive) DTDs. It is not clear whether the restriction to a non-recursive DTD could help to substantially reduce the complexity for Proposition 3.10. The complexity of satisfiability of VPA under DTD constraint is thus higher than the complexity of VPA satisfiability in general (at least, our upper bound is). This is because because VPA intersection, unlike $\mathcal{X}Reg$ intersection, may involve a quadratic size increase. Therefore, adding a constraint in the form of an external DTD raises the complexity in spite of the fact that every DTD can be converted to an equivalent VPA of linear size. We have already observed that evaluation of *FO* or *MSO* formulae is in PSPACE. Satisfiability of such formulae, however, is non-elementary [Sto74].

The table in Figure 3.10 summarizes the complexity of evaluation and satisfiability for Boolean queries. Queries are denoted by Q (XPath dialects or logical queries), or $\mathcal{A}$ (automata), whereas the document is denoted by t. The results in red are, to the best of our knowledge, new contributions.

3.3.4. Tree Alignments, a Model for Queries, Views and Updates

Tree automata can clearly specify Boolean queries: an automaton $\mathcal{A}$ represents the Boolean query Q such that $t \models Q$ iff $t \in L(\mathcal{A})$. The connection between tree automata and unary queries is less straightforward. We settled upon tree alignments to represent queries and views.

Conceptually, views and queries are very similar objects, because our views simply select nodes which should be visible. Our views can also relabel nodes, but since we assume a finite alphabet, the information regarding the relabeling can be managed similarly to the selection by a tree automaton. Views and updates both take as input a document, and output another document. On the other hand, views and updates have a different semantics: a view constructs a new document and keeps the original document unchanged, Nevertheless, this difference is a mere question of how we interpret the relation between the input and output document, and this difference does not raise any trouble at our level of modelization. Therefore, we will use the same formalism to reason about views and updates, and this formalism will also represent queries. Queries, views and updates will be represented as tree alignments, and we will add specific constraints for the tree alignments representing queries and views.

Tree Alignments Tree alignments represent k versions of a document as a single tree whose nodes are labeled with k-uples. Each component of this tree stands for one of the versions, and an ε symbol on the i^{th} component in the label of node n means that node n is not present in this version of the document. Given a natural k, we define the alphabet $\Sigma_{\text{edit},k}$ as $\Sigma_{\text{edit},k} = \Sigma^k \backslash \{(\varepsilon, \ldots, \varepsilon)\}$. For the special binary case ($k = 2$), we drop the subscript and write Σ_{edit}. We also use Σ_ε to denote the alphabet $\Sigma \cup \{\varepsilon\}$.

A k, Σ-alignment – or *alignment* for short – is a tree t over $\Sigma_{\text{edit},k}$, such that $lab_t(root_t) = (r, \ldots, r)$ for some $r \in \Sigma$. The alignment is *upward-closed* if, for every natural i and every node $n \in N_t$ such that the i^{th} component of $lab_t(n)$ is ε, the i^{th} component of $lab_t(n')$ is also ε for every descendant n' of n. An upward-closed k, Σ-alignment for $k = 2$ is called an *editing script*. For every alphabet Σ and naturals $i \leqslant k$, the projection π_i^k over the i^{th} component is the morphism that maps $(a_1, \ldots, a_k)$ into a_i. We extend the definition of projection in order to manage several component: given m integers $i_1, i_2 \ldots, i_m$ in $\{1, \ldots, k\}$, the projection $\pi^k_{i_1,\ldots,i_m}$ is the morphism that maps $(a_1, \ldots, a_k)$ into $(a_{i_1}, \ldots, a_{i_m})$. We drop the superscript whenever it is not relevant, and write π_i instead of π_i^k, or $\pi_{i_1,\ldots,i_m}$ instead of $\pi^k_{i_1,\ldots,i_m}$. Note that by Proposition 3.3, we get directly:

Proposition 3.18. *Any projection of a regular set of upward-closed alignments is also a regular set of alignments.*

Maximal Languages, and Queries as Tree Alignments A set of tree alignments L is *maximal* if for every t, t' in L, $\pi_1(t) = \pi_1(t')$ implies $t = t'$. We will essentially use maximal languages for the representation of queries and views.

Example 3.6. *The set of tree alignments in Figure 3.11 is maximal because none of the three trees t_1, t_2 and t_3 have the same projection on the first component. This set would still be maximal if the node n_3 in t_3 was labeled (a, b) because $\pi_1(t_1)$ and $\pi_1(t_3)$, though isomorphic, would still not be equal.*

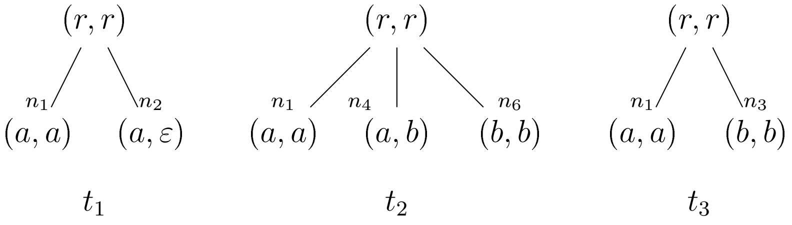


Figure 3.11.: A maximal set of tree alignments

On the other hand, if the rightmost node in t_3 was labeled (a, b) and had identifier n_2 instead of n_3, then the language $\{t_1, t_2, t_3\}$ would not be maximal.

Notation. *Given maximal sets of 2-alignments Q_1 and Q_2 over Σ, and a tree $t \in \pi_1(Q_1) \cap \pi_1(Q_2)$, we denote by $t \otimes Q_1$ the unique tree alignment $t_0 \in Q_1$ such that $t = \pi_1(t_0)$. We also denote by $t \otimes Q_1 \otimes Q_2$ the unique $3, \Sigma$ alignment t_1 such that $\pi_{1,2}(t_1) \in Q_1$, $\pi_{1,3}(t_1) \in Q_2$, and $t = \pi_1(t_1)$. Given two maximal languages Q_1 and Q_2 and a languages L over Σ, we denote by $L_{Q_1 \otimes Q_2}$ the set of trees $\{t \otimes Q_1 \otimes Q_2 \mid t \in L\}$.*

Remark 3.6. *With the notations of Chapter 5, $t \otimes Q_1 = \pi_{1,1}(t) \circ Q_1$ and $t \otimes Q_1 \otimes Q_2 = \pi_{2,1,4}(Q_1 \bowtie \pi_{1,1}(t) \bowtie Q_2)$.*

These notations will be used to represent the nodes selected by a query on some tree: to each query Q_1 is naturally associated a set of tree alignments L over $\Sigma \times \Sigma_\varepsilon$, such that for every $t \in L$ and $n \in N_t$, $lab_t(n) \in \{(a, a) \mid a \in \Sigma\}$ if $n \in Q_1(t)$, and $lab_t(n) \in \Sigma \times \{\varepsilon\}$ otherwise. We will sometimes identify the query Q_1 with its representation L via tree alignments. Given two queries Q_1 and Q_2 and a tree t, the tree $t \otimes Q_1 \otimes Q_2$ allows to represent simultaneously the nodes selected by Q_1 and Q_2 over t.

Example 3.7. *The tree in Figure 3.12 is $t_0 \otimes Q_{\mathcal{X}}$, where $Q_{\mathcal{X}}$ is the query from Example 3.5. This is an upward-closed alignment.*

Query Automata A *query automaton* is an automaton $\mathcal{A}$ that only accepts trees over $\Sigma_{\text{query}} = \{(a, a) \mid a \in \Sigma\} \cup \{(a, \varepsilon) \mid a \in \Sigma\}$, such that $L(\mathcal{A})$ is a maximal language. Every query automaton $\mathcal{A}$ represents a query $Q_{\mathcal{A}}$ defined as follows: for every tree t, if $t \notin \pi_1(L(\mathcal{A}))$ then $Q_{\mathcal{A}}(t) = \varnothing$ otherwise $t \otimes L(\mathcal{A})$ defines a tree t' over Σ_{query} and $Q_{\mathcal{A}}(t) = \{n \in N_t \mid lab_{t'}(n) \notin \Sigma \times \{\varepsilon\}\}$.

We extend query automata to allow relabeling: a *view automaton* is an automaton $\mathcal{A}$ that only accepts trees over $\Sigma \times \Sigma_\varepsilon$, such that $L(\mathcal{A})$ is a maximal language.

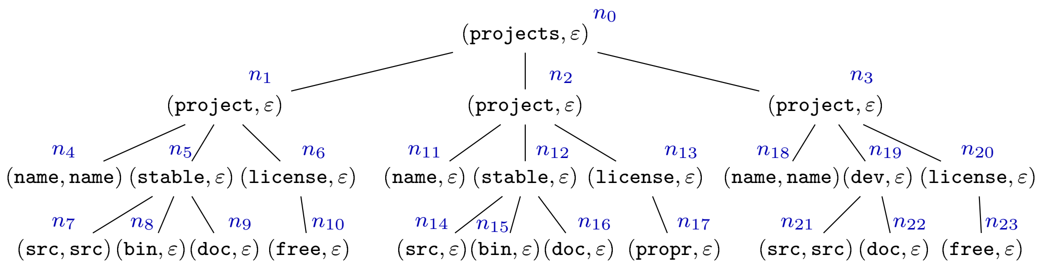


Figure 3.12.: Tree alignment $t_0 \otimes Q_{\mathcal{X}}$

Views Queries can only select nodes, according to our definition. Views, however, should additionally allow relabeling. In general, the purpose of a view is to map a document to another document that will be provided to the user. We can consider a view as a mapping V that takes as input a document t and a node $n \in N_t$, and outputs $V(t, n) \in \Sigma \cup \{\varepsilon\}$, the label of n in the view (or ε if n is hidden by the view).

Definition 3.3. *A* view *is a regular language V such that $V \subseteq T_{\Sigma\times\Sigma_\varepsilon}$ and V is a maximal language.*

Views are assumed to be closed under isomorphism like queries. The definition of views restricts the views to regular tree languages, so that every view can be represented by a view automaton. But in many parts of this dissertation, *views* will assume a more restrictive definition, or use another representation. In particular, the view will sometimes be specified not by an automaton, but by Regular XPath queries, or by schema-aware specifications. Also, we often use non-relabeling views to keep proofs simpler. Those restrictions or representations will be announced at the beginning of the corresponding sections.

Let V a view and $t \in \pi_1(t)$. Set t_1 the unique tree in V such that $\pi_1(t_1) = t$. A node $n \in N_t$ is *visible* w.r.t. view V if $lab_{t_1}(n)$ belongs to Σ^2, otherwise ($lab_{t_1}(n) \in \Sigma \times \{\varepsilon\}$) n is *hidden*. When $lab_{t_1}(n) = (a, b) \in \Sigma^2$, view V intuitively relabels the a-labeled node n into b. The *view tree* of t w.r.t. view V is the tree $\mathcal{V}iew\,(V, t)$ obtained from t by removing the nodes hidden by the view. Visible nodes whose parent is hidden are "adopted" by their closest visible ancestor. Formally, $\mathcal{V}iew\,(V, t)$ is defined as the tree $t' = (\Sigma, N_{t'}, child_{t'}, follow_{t'}, lab_{t'})$ with $N_{t'} = \{n \in N_t \mid lab_{t_1}(n) \notin \Sigma \times \{\varepsilon\}\}$, and, for every n labeled (a, b) in $N_{t'}$, $lab_{t'}(n) = b$, while the relations $child_{t'}$ and $follow_{t'}$ are defined as follows. The child relation of t' is best defined in terms of the ancestor relation $\preccurlyeq_{t'} = (\preccurlyeq_t \cap (N_{t'})^2)$: $child_{t'}$ is the set of all pairs $(n, n') \in N_{t'}$ such that $n \preccurlyeq_{t'} n'$ and such that there exist no n'' satisfying $n \preccurlyeq_{t'} n'' \preccurlyeq_{t'} n'$ apart from n and n'. The following sibling relation $follow_{t'}$

is defined as the set of all pairs (n, n') such that $Parent_{t'}(n) = Parent_{t'}(n')$ and n comes before n' in document order.

The *domain* of a view is $\pi_1(V)$, the set of all documents that admit some view tree w.r.t. V.

Each (root-preserving) query represents a view that does not relabel nodes; the nodes selected by the query are visible to the user, and those that are not are hidden from the user. We define the *view tree* of a tree t with respect to a given query Q: this view tree $\mathcal{V}iew\,(Q, t)$ is obtained from t by removing the nodes that are not selected by the query. Selected nodes whose parent is not selected are "adopted" by their closest selected ancestor. Formally, $\mathcal{V}iew\,(Q, t)$ is defined as the tree $t' = (\Sigma, N_{t'}, child_{t'}, follow_{t'}, lab_{t'})$ with $N_{t'} = Q(t)$, with $lab_{t'}(n) = lab_t(n)$ for every $n \in N_{t'}$, and with relations $child_{t'}$ and $follow_{t'}$ defined exactly as above. If we materialize the view, it is the document $\mathcal{V}iew\,(Q, t)$ that should be returned to the user. In the non-materialized setting, the user is not provided with $\mathcal{V}iew\,(Q, t)$ directly, but is allowed to pose queries on this view (possibly including the query: "return all nodes", which then will return $\mathcal{V}iew\,(Q, t)$). In the non-materialized setting, the user should be provided with a view schema, and the view schema for a view V with domain D should be a representation for $\mathcal{V}iew\,(Q, D) = \pi_2(V)$, as discussed in Chapter 6.

Notation. *We will often identify an automaton with the query or view it represents. Therefore we can use notations such as $\mathcal{V}iew\,(\mathcal{A}, t)$ where $\mathcal{A}$ is an automaton, or attribute to the automaton some property of its query, for instance speaking of root preserving view automata.*

All the views we consider keep the label of the root as is, so we will not even mention the fact that our view automata are root preserving. What is more, in our examples, we will sometimes specify non-relabeling views using XPath expressions. To keep shorter expressions, the expressions we use do not explicitly select the root of the tree, but it is implicitly assumed that the query does select the root.

Other Formalisms for Querying with Automata Several representations of unary queries through tree automata have been investigated. One possible representation consists in the extension of deterministic two-way automata with selecting states [NS02]. This model of automata (which we do not detail) is based on the two-way automata from [BKMW01]: actually, Neven and Schwentick prove that the two-way automata from [BKMW01] extended with selecting states have to be enhanced with special transitions in order to capture the expressiveness of *MSO* queries with one free variable. The original two-way automata and the enhanced ones thus accept the same language but do not compute the same queries. This property generalizes the observation that NTAs extended with selecting states loose their *MSO* expressivity if we require them to be deterministic. Frick et al. also extend classical automata models with selecting states [FGK03]. They propose to use NTAs (over binary encoding) extended with selecting states. On the one hand they give up determinism, but on the other hand they avoid the complexity of two-way automata models. Standard tree automata with selecting states are also used in [LS08, LS10, FDL11], but those are hedge automata, running over unranked trees instead of the binary encoding. Basically, a query automaton with selecting state is an NTA $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ extended with a subset S of Q that specifies the selecting states. Given a tree t, the nodes selected by $(\mathcal{A}, S)$ over t (under the existential semantics) are all nodes $n \in N_t$ such that there exists some accepting run ρ of $\mathcal{A}$ over t that maps n to a state in S. The expressive power of the model remains the same (namely, *MSO* unary queries) if the universal semantics is adopted instead of the existential semantics [FGK03], i.e., if n would be selected when *all* accepting runs ρ of $\mathcal{A}$ over t map n to states in S. Neither existential nor universal semantics are really convenient for reasoning tasks [LS08, LS10], not only because several runs need to be taken into account, but also because choosing either of those semantics makes operations such as complementation more complex.

To circumvent the limitations of the existential and universal semantics, Libkin and Sirangelo [LS08, LS10] propose to restrict the automata to single-run automata, for which the existential and universal semantics coincide: the automaton with selecting states $(\mathcal{A}, S)$ is *single run* if it accepts every tree and if, for every tree t and every pair ρ_1, ρ_2 of accepting runs over t, the nodes selected by ρ_1 and ρ_2 are the same. In a nutshell, since one cannot require determinism in the transitions, one merely asks the mapping from the tree to the set of selected nodes to be definable from a single run.

The *MSO* expressiveness of single run (and even unambiguous) query automata was already established in the litterature under different formalisms, such as the IBAGs of [NdB02]. Those results were generalized to n-ary queries by [NPTT05]: existential and universal semantics still have *MSO* expressiveness for n-ary queries, but unambiguous tree automata are strictly less expressive: they have exactly the expressive power of Boolean combinations of unary *MSO* queries.

Our model based on tree alignments also resorts to non-determinism, even when we use deterministic automata in order to define the regular set of tree alignments. In the tree-alignment model, the non-determinism lies in the choice of the second component of the tree: given a tree t and query automaton $\mathcal{A}$, if we wish to compute $Q_{\mathcal{A}}(t)$, we first need to guess the tree $t' \in L(\mathcal{A})$ such that $\pi_1(t') = t$. Only after we have guessed t' can we run the automaton $\mathcal{A}$ and check $t' \in L(\mathcal{A})$.

Updates In the chapter dealing with updates, we only consider upward-closed tree alignments. An update is then formalized as an editing script.

Each editing script t represents an update that takes as input $\pi_1(t)$ and outputs $\pi_2(t)$. This representation allows to identify not only the original document and the resulting document, but also the correspondence between the nodes of those trees. A node of the original document is deleted if it has label (a, ε) in t, preserved unchanged if it has label (a, a), and relabeled if it has label (a, b) with $a \neq b$, for some $a, b \in \Sigma$. A node with label (ε, a) in t represents a new node that is inserted in the resulting document. Our hypothesis regarding the label of the root in tree alignments implies that the root of the document is always preserved, while our restriction to editing scripts (upward-closed 2-alignments) implies that insertions and deletions can only involve whole subtrees and cannot be limited to internal nodes.

Alternative Transformation Languages The litterature presents a huge collection of tree transducer models: bottom-up and top-down tree transducers, macro tree transducers, attributed tree transducers, tree-walking transducers, visibly pushdown transducers, streaming tree transducers... Many of these were primarily designed for ranked trees, but have since been extended to unranked trees. Some of these models have also been enhanced with pebbles or lookahead mechanisms... The operations supported cover relabelings, insertions and deletions, copying or reordering of nodes and subtrees... The closest to our 2-alignments is the visibly pushdown transducer, that can manage unranked trees but does not allow to reorder or copy subtrees.

Visibly pushdown transducers extend VPAs with outputs. Several models of VPTs have been proposed [RS08, TVY08, FRR$^+$10], with different expressivity. Apart from [TVY08] those papers define visibly pushdown transducers that can express more than tree to tree transformations, as the pairs of corresponding opening and closing tag need not have the same label. To facilitate the comparison with our own transformation model, we survey which tree transformations can be expressed with these transducers, when we require the input and output to be the linearization of some trees.

The original versions of visibly pushdown transducers allowed the output of symbols without reading any input symbol [RS08, TVY08]. Each transition may read and output at most one symbol. When it reads an opening tag, it must output an opening tag, and similarly for closing tags. Transitions dealing with opening tags push a symbol onto the stack, and transitions dealing with closing tags pop a symbol. Raskin et al. [RS08] defines subfamilies of these visibly pushdown transducers that preserve regularity and have decidable typechecking. SVPTs partition the stack symbols into insertion, copy and deletion symbols, thus synchronizing the operations applied to the opening tag and its corresponding closing tag. Further restrictions of SVPTs through the exclusion of insertions, deletions or both yield the classes of SVPT$_{\mathrm{ni}}$, SVPT$_{\mathrm{nd}}$ and FSVPT. SVPT may match different occurrences of an opening tag (op, a) with different closing tags (e.g., (cl, b), (cl, c)), but the tree transformations they can express correspond to the regular sets of 2-alignments. FSVPTs can only relabel nodes (tags) and therefore the tree transformations they express correspond to the class of regular sets of alignments using only letters in Σ^2. SVPT$_{\mathrm{ni}}$ can only delete matching pairs of closing and opening tags and therefore the tree transformations they can express correspond to the class of regular sets of alignments using only letters in $\Sigma \times \Sigma_\varepsilon$.

In subsequent works, Filiot et al. [FRR+10, FGRS11] have adopted another definition for visibly pushdown transducers (VPTs), which map each transition of a VPA to a word over $\hat{\Sigma}$. This new model clearly subsumes FSVPTs and SVPT$_{\mathrm{ni}}$, without incurring the disadvantages of SVPT (and even SVPT$_{\mathrm{nd}}$) regarding decidability of functionality and equivalence. On the other hand, these VPTs are not closed under composition nor inverse, and the typechecking problem against a VPA is undecidable. *Well-nested VPTs* restrict these VPTs with a notion of synchronization between the opening and closing tags. Essentially, given any stack symbol γ, if there exist opening and closing transitions with stack symbol γ and respective output u_1 and u_2, then $u_1 u_2$ must be a well-nested word[8]. Servais presents in his Phd thesis [Ser11] a comprehensive survey of VPTs and well-nested VPTs together with a comparative analysis of the expressiveness of VPT and classical tree transducer models.

In contrast to our alignment representation, VPTs and well-nested VPTs do not offer a direct relationship between input nodes and output nodes. The increase in expressiveness obtained by the possibility to output several tags while reading a single input tag comes at this price, but on the other hand this notion of nodes is essentially relevant for XML document (or tree) manipulation, which was not the primary purpose of visibly pushdown machines. In terms of expressive power, the tree transformations definable by VPTs and well-nested VPTs are incomparable with regular sets of alignments and interval bounded regular sets of alignments. The only feature of interval bounded regular sets of alignments that cannot be handled by VPTs, however, are the unlimited insertions at the leaves, a rather cosmetic difference, whereas insertion of even one single internal node by 1-interval bounded alignments cannot be expressed by well-nested VPTs in general.

Example 3.8. *Let L the set of all alignments that take as input three arbitrary long threads of a, b and c below the root, that keep all nodes of the input unchanged and add a d node below the root as an ancestor of the two first threads, as illustrated on Figure 3.13. L is clearly a regular language of 1-interval bounded alignments, yet one cannot build a well-nested VPT*

[8] the authors define as well nested the smallest family comprising the empty word, and the concatenation of an opening tag with a well-nested word followed by a closing tag. $(op, a)\,(op, b)\,(cl, c)\,(cl, c)\,(op, a)\,(cl, b)$, for instance, is a well-nested word, though it is not the linearization of a tree

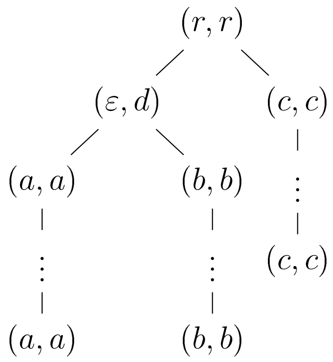


Figure 3.13.: A regular set of 1-interval bounded alignments

realizing the same transduction as L.

VPTs are only remotely related to regular sets of alignments, but the tree transformations they can express include the regular sets of alignments such that: (1) every insertion node has only insertion nodes as descendants or is the only child of its parent (2) the number of consecutive insertion nodes on any root-to-leaf path is bounded by a constant. Similarly to SVPTs, well-nested VPTs are closely related to regular sets of alignment. The image of a well-nested word by a well-nested VPT is still a nested word, and well-nested VPTs are closed under composition, though not under inverse. Well-nested VPTs have moreover decidable typechecking: given two VPAs A_1 and A_2 and a well-nested VPT T, the problem of checking if the image of A_1 by T is a subset of $L(A_2)$ is Exptime-complete, and is in PTime if A_2 is deterministic [FRR+10]. Let us consider the tree transformations that can be expressed by well-nested VPTs, i.e., we require the word u_1u_2 in the above description to be the linearization of some tree. Under this restriction, the well-nested VPTs model forms a strict subclass of regular sets of alignments but strictly generalizes $\mathrm{SVPT_{ni}}$.

3.3.5. XQUF

XQuery Update Facility (XQUF) [W3C09] is an extension of the XQuery language, for performing update operations on XML documents. It is based on the XQuery and XPath 2.0 data model, and is composed of non-updating expressions (classical XQuery) returning a result, and updating expressions returning nothing, like in SQL. It provides basic operations acting upon XML nodes:

- insert a (sequence of) node(s) after/before/as a children a specified node
- delete a (sequence of) node(s)

- rename a node without affecting its identity and content
- replace the children of a node with a sequence of nodes
- replace the value of a node by a string value

XQUF can copy all or parts of an XML document by iterated application of basic operations. It cannot move parts of a document into another part of the document, however, as justified on the XQuery-requirements page of the W3C [W3C11]: "A node can be deleted, and a copy inserted in a new location, but it will have a new identity. The Working Group felt that this functionality would limit the environments in which the XQuery Update Facility could be implemented."

Updating expressions are evaluated following the *snapshot* semantics: the query selects the node(s) to update, and describes the update operations to apply on those nodes; update operations are accumulated into a Pending Update List, and are executed all at once. Consider for instance the update query in Figure 3.14. It is irrelevant whether the `delete` is written after or before the `insert` operation. This query will insert an `a(b)` subtree before every node selected by path expression `/r/c[.//d]`, and delete all such nodes.

```
for $x in /r/c[.//d]
return
  delete $x ,
  insert a(b) before $x
```

Figure 3.14.: An update defined with XQUF.

From XQUF to Editing Scripts

We introduce a small fragment of XQUF that can be compiled into automata on editing scripts. That is, for every update query Q in that fragment, one can construct an automaton on editing scripts A_Q representing Q. For example Figure 3.15 shows an editing script u belonging to the language $L(A_Q)$ accepted by the automaton A_Q obtained from the XQUF query Q of Figure 3.14. More formally, let L_Q be the set of all editing scripts u such that $\pi_1(u)$ is transformed into $\pi_2(u)$ by Q. The editing scripts equivalent to editing scripts in the language $L(A_Q)$ form exactly the set L_Q. This XQUF fragment and the translation have limited expressiveness and efficiency, and are therefore rather intended as a proof-of-concept.

A Proof-of-concept Fragment of XQUF The following grammar defines the fragment of XQUF we are interested in. It essentially allows the aforementioned basic operations, and uses `for` expressions to identify nodes to be updated. The `for` expressions cannot be nested, and this is potentially the most severe restriction of our fragment.

$$
\begin{array}{rcl}
\text{Expr} & ::= & \text{SingleExpr } [\texttt{,}\ \text{SingleExpr}]^* \\
\text{SingleExpr} & ::= & \text{IfExpr} \mid \text{ForExpr} \mid \text{UpdateExpr} \\
\text{IfExpr} & ::= & \texttt{if (} \mathit{AbsolutePath} \texttt{)} \\
 & & \texttt{then}\ \text{Expr}\ \texttt{else}\ \text{Expr} \\
\text{ForExpr} & ::= & \texttt{for}\ \$\mathit{VarName}\ \texttt{in}\ \mathit{AbsolutePath} \\
 & & \texttt{return}\ \ \text{UpdateExpr} \\
\text{UpdateExpr} & ::= & \text{SingleUpdate } [\texttt{,}\ \text{SingleUpdate}]^* \\
\text{SingleUpdate} & ::= & \text{Insert} \mid \text{Delete} \mid \text{Rename} \mid \text{Replace} \\
\text{Insert} & ::= & \texttt{insert}\ \text{Source} \\
 & & [[[\texttt{as}[\texttt{first}|\texttt{last}]]?\texttt{into}]|\texttt{after}|\texttt{before}] \\
 & & \text{Target} \\
\text{Delete} & ::= & \texttt{delete}\ \text{Target} \\
\text{Rename} & ::= & \texttt{rename node}\ \text{Target}\ \texttt{as}\ \mathit{ElementName} \\
\text{Replace} & ::= & \texttt{replace node}\ \text{Target}\ \texttt{with}\ \text{Source} \\
\text{Target} & ::= & \$\mathit{VarName} \mid\ \mathit{AbsolutePath} \\
\text{Source} & ::= & \mathit{ConstantSequence}
\end{array}
$$

Here *AbsolutePath* means any NavXPath absolute path and *ConstantSequence* means any sequence of constant XML (sub)trees. Note that we basically distinguish two kinds of elementary update operations: those in which the target is specified by an XPath expression (e.g. `delete /a/b`; `insert a, b(a) as last into /r/c`), and those in which the target is specified by a variable (e.g. `rename node $var as e`; `replace node $var with a, b(), a`). The latter ones are to be used in the body of `for` update instructions (ForExpr rule). An expression in the fragment we consider (Expr) is a sequence of single expressions that can be update instructions (insert, delete, rename or replace), or a `for` update instruction (ForExpr rule), or an `if` update instruction (IfExpr rule). The body of the `for` update instruction can contain only a sequence of basic update operations (no nested `for`. The `if` expression can contain, in its then and else parts, any expression. Naturally, the target of basic update operations may be specified by a variable only when the variable is bound by a `for` expression.

Translation to Automata We only give here an intuition of how the translation works. We know from e.g., [CGLV09], that for every XPath query p, we can compute in exponential time an automaton B_p that accepts all trees t over $\Sigma \times \{0,1\}$ such that for every node $n \in u$, $lab_t(n) \in \Sigma \times \{1\}$ if and only if n is selected by p over $\pi_1(t)$.

Let Q be an XQUF update from the above fragment. We are going to decorate the editing scripts with intermediate results that represent the evaluation of the (absolute) path queries occurring in Q. Let $p_1, p_2, \ldots p_k$ be the absolute path queries appearing in Q.

First, we replace in Q every ForExpr expression `for` $\$x$ `in` p_i `return` U with update expression U in which we replace each occurrence of x with p_i. Technically speaking, the new expression is not exactly an XQUF update because in queries of the form `rename node` Target `as` *ElementName*, for instance, Target should evaluate into a single node. However, the intended semantics of such an extended expression is quite clear. Thanks to this transformation, we can suppose there are no variables in the SingleUpdate expressions of Q for the following construction. Let $u_1, u_2, \ldots u_m$ be the SingleUpdate expressions in (transformed) update Q. For each u_i, we clearly can build an automaton B_{u_i} with the required property $L(B_{u_i}) = L_{u_i}$.

Decorations for trees: Let Σ_{deco} be the alphabet built from $\Sigma \times \Sigma_\varepsilon \times \{0,1\}^k \times (\Sigma_\varepsilon)^m \cup \{\varepsilon\} \times \Sigma \times \{\varepsilon\}^{k+1} \times (\Sigma_\varepsilon)^m$ with the additional restriction that for every $\alpha \in \Sigma_{deco}$, if $\pi_1(\alpha) = \varepsilon$ then there is at most one $i \in \{k+3, \ldots k+m+2\}$ such that $\pi_i(\alpha) \neq \pi_1(\alpha)$. Intuitively, the first two components will represent the final XQuery update, the k next components represent the nodes selected by the k XPath queries $p_1, \ldots p_k$, and the last m components the result of the SingleUpdates $u_1, \ldots u_m$. We now define D as the (regular) set of all trees t over Σ_{deco} such that
(1) for every $i \in \{3, \ldots, k+2\}$, $\pi_{1,i}(t) \in L(B_{p_i})$ and
(2) for every $i \in \{k+3, \ldots, k+m+2\}$, $\pi_{1,i}(t) \in L(B_{u_i})$.

Building the automaton: We can build by induction an automaton A_Q over D such that $\pi_{1,2}(L(A_Q)) = L_Q$. We just sketch the construction for a single update and an `if` expression.

(SingleUpdate): For a single update u_i, then A_{u_i} selects the trees from D such that $\pi_2(t) = \pi_{(k+2+i)}(t)$, so this case is trivial.

(IfExpr): Given a query q of the form `if (`p_i`) then` e_1 `else` e_2, suppose we have computed automata A_{e_1} and A_{e_2}. Then, given any tree t, automaton A_q tests whether there exists a node with label 1 on the $(2+i)^{th}$ component of t. If so, A_q runs automaton A_{e_1} on t, otherwise it runs automaton A_{e_2}. This concludes our sketch of proof.

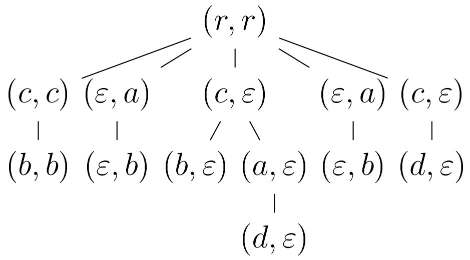


Figure 3.15.: An editing script from the XQUF query of Figure 3.14.

3.3.6. From Regular XPath to Automata

Calvanese et al. [CGLV09] define a linear algorithm converting Regular XPath Boolean expressions into two-way weak alternating automata (2-ATAs). They also provide an exponential algorithm converting Regular XPath expressions into NTAs. The conversion works as follows. First, they show how a 2-ATA $\mathcal{A}_\phi$ can be built in polynomial time from any Regular XPath Boolean filter ϕ, such that $L(\mathcal{A}_\phi) = \{fcns(t) \mid t \vDash \phi\}$. Then they show that for any 2-ATA $\mathcal{A}_\phi$ an NTA $\mathcal{A}$ equivalent to $\mathcal{A}_\phi$ can be built in exponential time from $\mathcal{A}_\phi$.

The construction of a VPA from a Regular XPath formula involves an exponential blowup. However, it is not always necessary to build the full automaton when we only wish to run the automaton on a given tree. We show how each transition can be tested using polynomial space only. This will be used in order to simulate this automaton in polynomial space on small trees (and particularly on trees of small depth). We copy from [CGLV09] their definition of 2-ATAs and their conversion from 2-ATAs to NTAs in order to prove the polynomial space simulation.

Two-way Weak Alternating Automata A 2-ATA is defined as a tuple $\mathcal{A} = (S, \Sigma, s_0, \delta, \alpha)$ where, to keep things simple [9], S is a finite set of states, $s_0 \in S$ is the initial state, Σ the alphabet, δ the set of transitions, and α the acceptance condition. The set of transitions δ is a function mapping pairs $(s, a) \in S \times \Sigma$ to positive Boolean formulae over $\{-1, 0, 1, 2\} \times S$, where positive Boolean formulae over I are defined inductively as $\phi \coloneqq x \in I \mid \mathsf{true} \mid \mathsf{false} \mid \phi \vee \phi \mid \phi \wedge \phi$. Let us introduce a notation that we use for the definition of runs: for every node n of t, we denote by $n.-1$ the parent of n, by $n.0$ the node n itself, by $n.1$ the left child of n, and by $n.2$ the right child of n, if they exist.

A *run* of $\mathcal{A}$ over tree t from node $n \in N_t$ is a (possibly infinite) tree t' over alphabet $\Sigma_{t'} = N_t \times S$, verifying the following two properties: (1) $lab(root_{t'}) = (n, q_i)$, and (2) for every node $n' \in N_{t'}$ with $lab_{t'}(n') = (x, q)$, there exist $k \geqslant 0$ and a set $S' = \{(\beta_1, s_1), \dots (\beta_k, s_k)\} \subseteq \{-1, 0, 1, 2\} \times S$ satisfying $\delta(q, lab_t(x))$ and such that for every $(\beta_i, s_i) \in S'$, $x.\beta_i$ is a node of t and n' has a child in t' with label $(x.\beta_i, s_i)$.

A path in t' is a (possibly infinite) sequence $P = n_0, n_1, \dots$ of nodes from t' such that $n_0 = root_{t'}$ and for every n_i in P $(i > 0)$, n_i is a child of n_{i-1}. The acceptance condition α partitions S into disjoints subsets $S_1, \dots, S_l$, some of which are accepting subsets, the others being rejecting subsets. Abusing the definition, we say that state s belongs to α if the component of s is accepting. Furthermore, those subsets follow a partial order $\leqslant_\alpha$ such that for every $i, j \leqslant l$, every $s, a \in S \times \Sigma$, and every state s' appearing in $\delta(s, a)$ with $s \in S_i$ and $s' \in S_j$, the component of s' is smaller than (or equal to)

[9] We restrict the original definition of 2-ATAs to binary trees

the component of s: $S_j \leqslant_\alpha S_i$. This particular ordering accounts for the denomination of "weak" automaton. Clearly, every infinite path P of t' eventually remains in some component: there exists a component $S(P) = S_i$ and some $k \in \mathbb{N}$ such that for every $k' > k$, the state q appearing in $lab_{t'}(n_{k'})$ belongs to S_i. A run is *accepting* if and only if for every infinite path P of t', the states of P end up in an accepting component, i.e., $S(P)$ is accepting. The language $L(\mathcal{A})$ of the 2-ATA $\mathcal{A}$ is the set of all trees t such that $\mathcal{A}$ has an accepting run from the root of t. This concludes the definition of 2-ATAs.

We wish to build from a given Regular XPath expression an equivalent 2-ATA on the *fcns* encoding. Yet one can build no 2-ATA on the *fcns* encoding that would have an accepting run from node n if and only if n is the first child of its parent. To remedy this shortcoming of the *fcns* encoding, Calvanese et al. enrich the node labels with four new tags: $\{ifc, irs, hfc, hrs\}$, as discussed on page 68. Given an unranked tree t, we denote by t^{deco} the decorated version of $fcns(t)$.

Theorem 3.19 (Theorem[CGLV09]). *Given a Regular XPath formula ϕ, one can build in polynomial time a 2-ATA $\mathcal{A}_\phi$ such that for every (unranked) tree t, $\mathcal{A}_\phi$ accepts t^{deco} if and only if $t \vDash \phi$.*

We do not detail their algorithm and refer the reader to [CGLV09] instead. The complexity of the algorithm is actually linear: Calvanese et al. observe that the number of states in $\mathcal{A}_\phi$ is linear. From their algorithm it is also clear that the number of transition rules is also linear: at some point in their algorithm they build word automata to represent path expressions, but using NFAs with ϵ-transitions guarantees linear complexity.

Conversion from 2-ATA into NTAs Let $\mathcal{A} = (S, \Sigma, s_0, \delta, \alpha)$ a 2-ATA. Calvanese et al. outline in [CGLV09] a construction from [Var98] that builds in exponential time an NTA $\mathcal{A}_n$ with the same language as $\mathcal{A}$.

Let $\mathcal{A}_n = (\Sigma, Q, Q_f, \Delta)$ be defined as follows. The set of states is $Q = \mathcal{P}(S) \times \mathcal{P}(S \times S) \times \mathcal{P}(S \times S)$, and the final states are $Q_f = \{(S_0, \tau_0, \eta_0) \mid s_0 \in S_0\}$. Automaton $\mathcal{A}_n$ has transition $a((S_1, \tau_1, \eta_1), (S_2, \tau_2, \eta_2)) \rightarrow (S_0, \tau_0, \eta_0)$ if and only if there exists some set $\mathcal{E} \subseteq S_0 \times \{0,1,2\} \times S$ such that, if we denote by θ_0 the union $\theta_0 = \mathcal{E} \cup \{(s, -1, s') \mid (s, s') \in \tau_0\}$, the following 5 conditions are satisfied:

1. for every $s \in S_0$, $\{(\beta, s') \mid (s, \beta, s') \in \theta_0\}$ satisfies $\delta(s, a)$
2. for every $\beta \in \{0,1,2\}$, $\{s' \mid \exists s \in S_0.(s, \beta, s') \in \theta_0\} \subseteq S_\beta$ and similarly for every $\beta \in \{1,2\}$, $\{s'' \mid \exists s' \in S_\beta.(s', s'') \in \tau_\beta\} \subseteq S_0$.
3. for every $(s, 0, s') \in \theta_0$, $(s, s') \in \eta_0$.

4. for every $\beta \in \{1, 2\}$, if $(s, \beta, s') \in \theta_0$, $(s', s'') \in \eta_\beta$ and $(s'', s''') \in \tau_\beta$, then $(s, s''') \in \eta_0$. Similarly, if $(s, s') \in \tau_\beta$, $(s', s'') \in \eta_0$ and $(s'', \beta, s''') \in \theta_0$, then $(s, s''') \in \eta_\beta$.

5. η_0 is closed under transitive closure: $(s, s'), (s', s'') \in \eta_0 \implies (s, s'') \in \eta_0$,

6. and for every $s \in S$ such that $(s, s) \in \eta_i$, then s belongs to α.

Intuitively, the first component S_i represents the set of all states that are mapped to the node. The second component τ_i represents the strategy applied. More accurately, θ_0 represents the actual strategy, and τ_0 is the subset corresponding to the upward moves of θ_0. Storing the other moves is not necessary as they can be guessed non-deterministically inside the transitions. Finally, the third component η_i represents the annotation.[10] This third component may actually be larger than the annotation corresponding to the chosen strategy θ_0, but the larger η is, the more difficult it is to obtain an accepting run of $\mathcal{A}_n$. The first two conditions check that the strategy satisfies the transitions of the alternating automaton, and that the set of current states is correct w.r.t. the strategy. The last three rules check that η contains all the possible loops when the strategy (partially) defined by θ_0 and the τ_i is applied. The last rule additionally checks that the annotation is accepting: every infinite path must visit infinitely often in the same state s some node of the tree. Due to the weak acceptance condition, this infinite path is accepting if and only if state s belongs to α, because then every state appearing on that path between two occurrences of s also belongs to α.

We also define the rules that apply at the leaves. Automaton $\mathcal{A}_n$ has transition $\bot \rightarrow (S_0, \tau_0, \eta_0)$ if and only if there exists some set $\mathcal{E} \subseteq S_0 \times \{0\} \times S$ such that, if we denote by θ_0 the union $\theta_0 = \mathcal{E} \cup \{(s, -1, s') \mid (s, s') \in \tau_0\}$, the following 3 conditions are satisfied:

1. for every $s \in S_0$, $\{(\beta, s') \mid (s, \beta, s') \in \theta_0\}$ satisfies $\delta(s, \bot)$

2. for every $(s, 0, s') \in \theta_0$, $(s, s') \in \eta_0$.

3. η_0 is closed under transitive closure: $(s, s'), (s', s'') \in \eta_0 \implies (s, s'') \in \eta_0$, and for every $s \in S$ such that $(s, s) \in \eta_i$, then s belongs to α.

This concludes the description of the conversion from 2-ATAs into NTAs. Vardi [Var98] presents a slightly more general construction for the conversion of general (instead of weak) two-way alternating parity automata: there is no partial order on the components of the acceptance condition, and so the annotations constructed in [Var98] record the components visited within the loops. Another adaptation of the general construction to weak alternating automata is sketched in [CGLV09]. They obtain an NTA with the same

[10] see [CGLV09, Var98] for more details about strategy and annotation

number of states as we do, but inferring the states and transitions from their sketch of proof is not an easy task, because their exposition of annotations (loops) does not fully take benefit from the weak accepting conditions to remove the visited components.

Theorem 3.20 ([Var98, CGLV09]). *Let $\mathcal{A}$ a two-way weak alternating parity automaton (2-ATA) with n states. We can build in exponential time an NTA $\mathcal{A}_n$ such that $L(\mathcal{A}_n) = L(\mathcal{A})$. The resulting automaton $\mathcal{A}_n$ has 2^{2n^2+n} states.*

We are not aware of better bounds on the number of states required by an NTA to simulate a 2-ATA, nor of a non-trivial lower bound. One can achieve at least tiny improvements to the 2^{2n^2+n} bound: first, only the loops containing two states from the same rejecting component must be stored in the annotations (and there cannot be such a loop of the form (s, s)), so that annotations are actually partial orders over S. There is no simple formula for the number p_n of partial orders over a set of n elements, but this number can be estimated asymptotically, and is bounded by $2^{n^2/4+\Theta(n)}$ [BPS96]. This only lowers the number of states of $\mathcal{A}_n$ to $2^{5n^2/4+\Theta(n)}$ because the "strategy" component multiplies this with 2^{n^2}.

If we analyse the construction above, we observe the following property, which essentially says that the NTA obtained from converting the 2-ATA can be simulated on the fly without constructing the NTA explicitly. Actually the only property we exploit in this dissertation is that this transition can be checked in polynomial space.

Theorem 3.21. *Let $\mathcal{A}$ a 2-ATA, and $\mathcal{A}_n = (\Sigma, Q, Q_f, \Delta)$ the tree automaton (over fcns encoding) equivalent to $\mathcal{A}$, defined as above. Given an input consisting in $q, q_1, q_2 \in Q$, $a \in \Sigma$, and $\mathcal{A}$, [a] we can decide if (q, a, q_1, q_2) belongs to Δ in polynomial time.*

[a] note that $\mathcal{A}_n$ is not part of the input

Proof. We fix the notations: $q_0 = (S_0, \tau_0, \eta_0)$, $q_1 = (S_1, \tau_1, \eta_1)$ and $q_2 = (S_2, \tau_2, \eta_2)$. If we are satisfied with non-deterministic polynomial time, the result is trivial: we only need to guess the right set $\mathcal{E}$ then it is easy to check all conditions in polynomial time. To obtain a polynomial algorithm, we observe that the transitions of the 2-ATA which have to be satisfied in condition 1. are positive Boolean formulae. This implies that maximizing $\mathcal{E}$ only makes satisfaction of condition 1. easier, provided all other conditions are satisfied.

Set $\mathcal{E} = \mathcal{E}_0 \cup \mathcal{E}_1 \cup \mathcal{E}_2$, with $\mathcal{E}_0 = \{(s, 0, s') \mid (s, s') \in \eta_0\}$, and for every $i \in \{1, 2\}$, $\mathcal{E}_i$ contains exactly all tuples (s, i, s') such that the following four conditions are satisfied: (1) $s \in S_0$, (2) $s' \in S_i$, and (3) for all (s', s'') in η_i and all (s'', s''') in τ_i (if any), (s, s''') belongs to η_0 (4) for all (s'', s''') in τ_i and all

(s''', s) in η_0 (if any), (s'', s') belongs to η_0. The set $\mathcal{E}$ can clearly be computed in polynomial time and there exists a set satisfying conditions 1. to 5. if and only if the set $\mathcal{E} = \mathcal{E}_0 \cup \mathcal{E}_1 \cup \mathcal{E}_2$ satisfies them. Then we check conditions 1. to 5. in polynomial time. Actually, only conditions 1. 2. and 5. still need to be checked. The overall complexity is even cubic in $|S_0| + |S_1| + |S_2|$ as the composition or transitive closure of binary relations over a set U can be obtained in time $O(|U|^3)$. □

Given a Regular XPath formula ϕ, let us denote by VPA(ϕ) the visibly pushdown automaton obtained from ϕ by composing the linear translation from ϕ into a 2-ATA $\mathcal{A}_\phi$ detailed in [CGLV09], then the exponential translation from the 2-ATA $\mathcal{A}_\phi$ into an NTA $\mathcal{A}_n$ over *fcns* encoding detailed above, and finally the translation of $\mathcal{A}_n$ into a VPA as described on page 69. We observe that each state or stack symbol of VPA(ϕ) can be represented in space linear in ϕ. As a corollary of Theorem 3.21 we obtain:

Proposition 3.22. *Let ϕ a Regular XPath formula, and Δ the set of transitions of VPA(ϕ), the VPA equivalent to ϕ obtained through the conversion of ϕ as detailed above. Given an input consisting in $q, q' \in Q$, $\gamma \in \Gamma$, $\eta \in \{op, cl\}$, $a \in \Sigma$, and ϕ* [11] *we can decide if (q, η, a, γ, q') belongs to Δ in polynomial time.*

Beyond [CGLV09], several authors study the translation of expressive XPath fragments to automata. We only survey some of the most recent constructions. Bjorklund et al. [BGM10] translate in linear time NavXPath formulae into loop-free two-way alternating tree automata over the binary *fcns* encoding. The resulting alternating automaton is in turn translated in exponential time into an usual NTA over the same encoding. The authors do not compare the construction to that of [CGLV09], but the translations in both papers seem quite similar, all the more so since the second phase relies in both cases on the original works by Vardi [Var98]. The essential difference lies in the lesser expressiveness of the XPath fragment considered in [BGM10], which results in simpler alternating automata: the runs of a loop-free two-way automaton are finite. Interestingly, the authors also observe that their translation provides an alternative to the translation of [LS10], although Libkin and Sirangelo translate the more expressive Conditional XPath extension of XPath.

Ten Cate and Lutz [tCL09] provide a translation for an even richer fragment of XPath: CoreXPath($*, \approx$), discussed on page 154. The purpose of this translation is also to prove that satisfiability for the corresponding XPath fragment has EXPTIME complexity, so the authors are only interested in the polynomial complexity of the translation to alternating automata, and the resulting two-way alternating automaton may

[11] note that VPA(ϕ) is not part of the input

have quadratically many states.

Libkin and Sirangelo [LS08, LS10] avoid both the intermediate step through two-way automata and the associated binary encoding of the document. They provide a translation of Conditional XPath into hedge automata through an intermediate representation in the temporal logic $\mathrm{TL}^{\mathrm{tree}}$. This translation involves only a single exponential. Francis et al. [FDL11] compute the same translation directly without resorting to the intermediate $\mathrm{TL}^{\mathrm{tree}}$ representation.

From XPath to deterministic automata: no way round the double exponential The exponential blowup in the conversion from a Boolean $\mathcal{X}Reg$ formula to any NTA cannot be avoided. However, when one requires deterministic automata, the determinization of NTAs involves yet another exponential blowup, so one could wonder if this doubly exponential blowup could be avoided in a direct translation from $\mathcal{X}Reg$ to deterministic automata. Kupferman et al. [KV05, KR10] prove it cannot.

Theorem 3.23 ([KR10]). *There exists an LTL formula ϕ of size $O(m)$ such that every DFA accepting $L(\phi)$ is of size doubly exponential in m.*

The proof uses essentially a language from [CKS81] based on the binary representation of naturals. There are 2^m different sequences of m bits, and the formula is devised so as to make sure that any equivalent deterministic automata needs to record an arbitrary subset of $\{1, \dots, 2^m\}$, which requires 2^{2^m} states. Let us recall the formula used in [KV05] with Regular XPath syntax in order to precise some fragment of Regular XPath that shows this blowup property. The formula used in [KV05] is essentially[12] $L'_m = \{(a+b+\#)^*\#w\#(a+b+\#)^*\$w \mid w \in \{a,b\}^m\}$. As every DFA accepting L'_m requires 2^{2^m} states, this language witnesses a doubly exponential blowup of the form $2^{2^{\Omega(\sqrt{m})}}$ from LTL to DFA, a result which has been subsequently improved by Kupferman et al. [KR10] to a $2^{2^{\Omega(m)}}$ lower bound when the size of the alphabet is not fixed and $2^{2^{\Omega(m/(\log m))}}$ for fixed-size alphabet. The formula for the unbounded alphabet is defined as follows. Let $\Sigma(m)$ denote the alphabet $\{a_1, \dots, a_m, b_1, \dots, b_m, \$, \#\}$ and r_m the expression $r_m = (a_1 + b_1)(a_2 + b_2) \dots (a_m + b_m)$. Let L_m denote the language $L_m = \{(r_m\#)^*w(\#r_m)^*\$w \mid w \in r_m\}$. Every DFA accepting L_m needs 2^{2^m} states. XPath however can represent L_m with the following formula ψ_m of size $O(m)$:

$$\psi_m = [\,\mathsf{not} \Leftarrow] \;\mathsf{and}\; (\phi_1) \;\mathsf{and}\; (\phi_2) \;\mathsf{and}\; \left(\Rightarrow^*::[\mathsf{self}::\$ \;\mathsf{and}\; \phi_0 \;\mathsf{and}\; \mathsf{not}(\Leftarrow^+::\$)]\right)$$

[12] The result in [KV05], deals with infinite words and deterministic Büchi automata, so they append an infinite sequence of # at the end of L_m. Since we only consider finite words, we stick (or revert to) to the original language from [CKS81] and stop after w. This does not affect the result.

where ϕ_0, ϕ_1, and ϕ_2 are as defined below:

$$\phi_0 = [\Rightarrow/[a_1 \text{ or } b_1]/\Rightarrow/[a_2 \text{ or } b_2]/\ldots/\Rightarrow/[a_m \text{ or } b_m]/[\text{not}\Rightarrow]]$$

$$\phi_1 = [a_1 \text{ or } b_1] \text{ and } [\text{not}[\Rightarrow^*/[\bigvee_{i\leqslant m+1} f_i]/\Rightarrow^*\$]]$$

$$\phi_2 = [\Rightarrow^*/[\text{self}::\# \text{ and } (\Rightarrow/\phi_3)^*/\Rightarrow::\#]]$$

The filter expression ϕ_3 is defined as follows:

$$\phi_3 = \bigvee_{i\leqslant m} ([a_i \wedge [\Rightarrow^*::\$/\Rightarrow^*::a_i]] \text{ or } [b_i \wedge [\Rightarrow^*::\$/\Rightarrow^*::b_i]])$$

For every $i \leqslant m-1$, f_i is defined as:

$$f_i = [a_i \text{ or } b_i] \text{ and } [\Rightarrow/[\text{not}(a_{i+1} \text{ or } b_{i+1})]]$$

and finally:

$$f_m = [a_m \text{ or } b_m] \text{ and } [\Rightarrow/[\text{not}(\# \text{ or } \$)]]$$

$$f_{m+1} = \text{self}::\# \text{ and } [\Rightarrow/[\text{not}(a_1 \text{ or } b_1)]]$$

In the formula above, ϕ_0 and ϕ_1 are used to check that the word belongs to $(r_m\#)^*r_m\$r_m$, and ϕ_2 checks that at some position before the \$, the next m symbols form the same word as the word w after the \$ because it enforces the satisfaction of ϕ_3 until the next $\#$. Thus, the family of formulae $(\psi_m)_{m\geqslant 1}$ witnesses a doubly exponential blowup $2^{2^{\Omega(m)}}$ from Regular XPath to DFAs.

Remark 3.7. *The language L'_m can be expressed with an LTL formula of size $O(m^2)$ [KV05]. This formula does not exploit the "until" operator of LTL and can therefore be expressed as a NavXPath formula, as presented in the appendix (p. 277).*

However, and unlike the formula in the appendix, the formulae ψ_m use the Kleene star of Regular XPath, in formula ϕ_2. We observe nevertheless that this Kleene star only covers a single axis with a filter, so that only Conditional XPath is used and not the full expressive power of Regular XPath. This of course is not surprising since the restriction of Conditional XPath to the horizontal axes corresponds to LTL (modulo linear translation).

This proof for the doubly exponential blowup from the translation of Regular XPath formulae into DFAs immediately implies the same blowup toward deterministic VPAs, and can be adapted to deterministic tree automata over the *fcns* encoding, etc. Also we have observed that the full expressive power of Regular XPath is not necessary: the formula in appendix shows that a doubly exponential blowup $2^{2^{\Omega(\sqrt{m})}}$ can already be observed, for instance, with the fragment of NavXPath containing only the vertical axes, their transitive closure, and filters (with negation): $(\Uparrow, \Uparrow^*, \Downarrow, \Downarrow^*, [\,], \neg)$.

The language L_m was used in [CKS81] to establish the doubly exponential blowup for the translation alternating automata to deterministic ones, and in [KV05, KR10] to prove the same gap from LTL to deterministic automata. But similar languages have also be used to show that some temporal logics are more succinct when past-time modalities are allowed. Benedikt and Jeffrey [BJ07] already use the language L_m in the XML setting to show that some XPath-like language cannot be evaluated in subexponential space.

4. XML Security Views: the Non-materialized Approach

Contents

The previous chapter surveyed tools and techniques that we use in this dissertation. This chapter focuses on the interaction of security views with read-only, unary queries. In the first section, we provide an XPath-based, user-friendly formalism for the specification of security views. We also describe the corresponding evaluation process for queries on non-materialized views, and investigate reasonable restrictions to facilitate the verification of policies by eliminating pathological view definitions. The second section is devoted to policy comparison. Using logical (automata-theoretic) methods we try to assess what information is disclosed by the views. We provide a few tools to verify security properties on the views, and evaluate the complexity of the resulting problems.

4.1. Specifying the Security Views

XPath expressions have been proposed as a fine-grained user-friendly formalisms for XML access control. Fan et al. propose to define the security view using XPath queries correlated to the DTD of the document. The simplest formalism could be to define the view through a single (Regular XPath) query that selects all visible nodes. Writing the query, however, could prove tedious and this approach departs from the spirit of traditional access control formalisms, where the possibility to write a list of simple rules, and the propagation of privileges (for instance from ancestors to descendants) are considered crucial features. We prove that in the case of DTDs annotated with Regular XPath expressions, both approaches can be reconciled. Indeed every security view defined by annotated DTDs can be expressed with a single Regular XPath query and vice-versa, a fact deemed impossible by some due to the "hierarchical structure and the dependency (e.g., ancestors and descendants) of XML data as well as the presence of disjunction and recursion in DTDs" [Ras07]. The justification for these seemingly conflicting statements probably lies in the higher expressiveness of Regular XPath queries w.r.t. the XPath fragment of [Ras07], which allows to support both DTDs and propagation rules for accessibility.

4.1.1. Annotated DTDs and Regular XPath

Following the approach in [FCG04, KMR05], we introduce *annotated DTDs* which consist in a pair (D, ann) with $D = (\Sigma, r, P)$ a DTD, and $\mathsf{ann} : \Sigma \times \Sigma \to \mathcal{X}Reg$ an annotation, i.e., a (partial) function mapping pairs of symbols to Regular XPath filter expressions.

The size of (D, ann) is $|D|$ plus the size of all filters f for every mapping $(a, b) \mapsto f$ in ann. The DTD D describes the schema of the document, whereas ann specifies the visibility of the nodes. Essentially, ann adds the security information to the production rules: the visibility of a node n labeled a and with parent labeled b is specified by filter $\mathsf{ann}(a, b)$. In case $\mathsf{ann}(a, b)$ is not explicitly defined by ann, a default policy is assumed: n inherits the visibility of its closest ancestor for which ann explicitly defines the visibility. The root is always assumed to be visible. An annotation is *simple* if it uses only the trivial filters true and false.

Annotated DTDs define views that do not relabel nodes, i.e., queries. We denote by $Q_{(D,\mathsf{ann})}$ the query defined by the annotated DTD (D, ann). Formally, given a document t, and $n \in N_t$, n belongs to $Q_{(D,\mathsf{ann})}(t)$ if and only if one of the following three conditions is satisfied: either (1) n is the root, or (2) $t, n \vDash \mathsf{ann}(lab_t(Parent_t(n)), lab_t(n))$, or (3) $Parent_t(n)$ belongs to $Q_{(D,\mathsf{ann})}(t)$ and $\mathsf{ann}(lab_t(Parent_t(n)), lab_t(n))$ is not specified. Annotated DTDs are useful to structure the specification of the policy. But in terms of expressiveness, they are equivalent to a single Regular XPath query:

Lemma 4.1. *For every annotated DTD* (D, ann)*, we can compute a* $\mathcal{X}Reg$ *filter* $\mathcal{X}^{\mathsf{ann}}_{\mathrm{acc}}$ *such that for every tree* $t \in D$ *and node* $n \in N_t$*,* $t, n \models \mathcal{X}^{\mathsf{ann}}_{\mathrm{acc}}$ *iff* n *is visible for* ann*. Similarly, the query* $Q_{(D,\mathsf{ann})}$ *can be expressed with a Regular XPath formula. Moreover, these filter and formula can be computed in linear time.*

Proof. The proof is rather straightforward; we use the filters from function ann and exploit the transitive closure from Regular XPath in order to simulate the inheritance. By $\mathrm{dom}(\mathsf{ann})$ we denote the set of pairs of symbols for which ann is defined. We begin by defining two filter expressions. The first checks if ann defines a filter expression for the current node

$$\mathcal{X}_{\mathrm{dom}} \coloneqq \bigvee\nolimits_{(a,b)\in\mathrm{dom}(\mathsf{ann})} \big(\mathsf{self}::b \ \mathsf{and}\ \Uparrow::a\big),$$

and if it is the case, the filter expression defined by ann is used to evaluate it

$$\mathcal{X}_{\mathrm{eval}} \coloneqq \bigvee\nolimits_{(a,b)\in\mathrm{dom}(\mathsf{ann})} \big(\mathsf{self}::b \ \mathsf{and}\ \Uparrow::a \ \mathsf{and}\ \mathsf{ann}(a,b)\big).$$

Finally, we restate the definition of accessibility using $\mathcal{X}Reg$

$$\mathcal{X}^{\mathsf{ann}}_{\mathrm{acc}} \coloneqq ([\,\mathsf{not}\,\mathcal{X}_{\mathrm{dom}}]/\Uparrow)^*/[\,\mathsf{not}(\Uparrow)\ \mathsf{or}\ \mathcal{X}_{\mathrm{eval}}]$$

Lemma 3.17 allows us to compute in time $O(|D|)$ a $\mathcal{X}Reg$ filter f_D equivalent to D. The query $Q_{(D,\mathsf{ann})}$ can be expressed with $[f_D]/\Downarrow^*[\mathcal{X}^{\mathsf{ann}}_{\mathrm{acc}}]$. □

In the following, we will investigate two different models for views: either the view will be given by a Regular XPath formula, or it will be given by a (maximal) regular set of tree alignments. We assume the source documents belong to a schema D, given by a DTD or an automaton over T_Σ. Of course, the domain may be T_Σ itself, if we do not want to constrain the possible source documents. This assumption is mainly used in section 4.2: when we compare two queries Q_1 and Q_2, we assume they both have same domain: $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, and only the nodes they select may vary.

Example 4.1. *The DTD* D_0 *below captures the schema of XML databases described in Example 3.1. We define here the annotated DTD* $A_0 = (D_0, \mathsf{ann}_0)$*.*

$\texttt{projects} \to \texttt{project}^*$
$\texttt{project} \to \texttt{name}, (\texttt{stable} \mid \texttt{dev}), \texttt{license}$
  $\mathsf{ann}_0(\texttt{project}, \texttt{stable}) = \mathsf{false}$
  $\mathsf{ann}_0(\texttt{project}, \texttt{dev}) = \mathsf{false}$
$\texttt{license} \to \texttt{free} \mid \texttt{propr}$

$\texttt{stable} \to \texttt{src}, \texttt{bin}, \texttt{doc}$
  $\mathsf{ann}_0(\texttt{stable}, \texttt{src}) = [\Uparrow^*::\texttt{project}/\Downarrow^*::\texttt{free}]$
  $\mathsf{ann}_0(\texttt{stable}, \texttt{doc}) = \mathsf{true}$
$\texttt{dev} \to \texttt{src}, \texttt{doc}$
  $\mathsf{ann}_0(\texttt{dev}, \texttt{src}) = [\Uparrow^*::\texttt{project}/\Downarrow^*::\texttt{free}]$
  $\mathsf{ann}_0(\texttt{dev}, \texttt{doc}) = \mathsf{true}$

The annotation ann_0 *gives access to all projects but hides the information whether or not the project is stable (in particular, it hides binaries). Additionally,* ann_0 *hides the source code of all projects developed under proprietary license.*

In the tree t_0 from Fig. 3.1 the root node `projects` *is accessible and all nodes* `project` *are accessible by inheritance. The nodes* `name` *and* `license` *with their children are accessible by inheritance as well.* ann_0 *implicitly states that* `stable` *and* `dev` *are not to be accessible, and the nodes* `bin` *are inaccessible by inheritance. On the other hand,* ann_0 *overrides the inheritance for nodes* `doc` *and makes them accessible. Finally, the accessibility of* `src` *nodes is conditional: only n_7 and n_{21} are accessible because only those satisfy the specified conditions,* $\mathsf{ann}_0(\mathtt{stable}, \mathtt{src})$ *and* $\mathsf{ann}_0(\mathtt{dev}, \mathtt{src})$ *resp. Figure 4.1 presents* $\mathcal{V}iew\left(Q_{(D_0,\mathsf{ann}_0)}, t_0\right)$ *for t_0 from Fig. 3.1.*

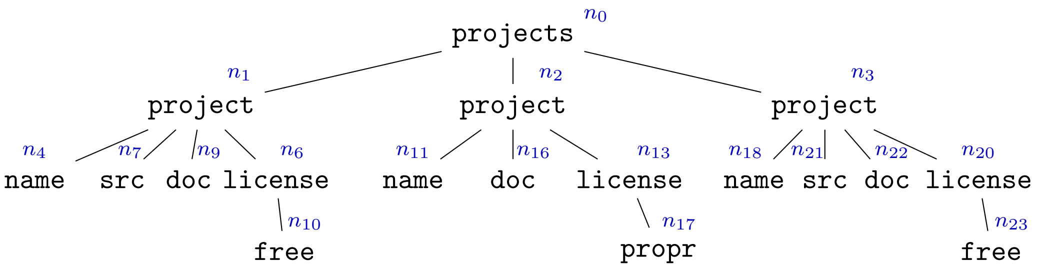


Figure 4.1.: The view $\mathcal{V}iew\left(Q_{(D_0,\mathsf{ann}_0)}, t_0\right)$.

4.1.2. Restrictions on the views

Defining the view with unrestricted Regular XPath queries or automata over tree alignments raises a major difficulty: the set of view trees $\mathcal{V}iew\,(V, D) = \{\mathcal{V}iew\,(V, t) \mid t \in D\}$ needs not be regular. Section 6.3 proposes solutions to compute a view schema in this case. Nonetheless, this non-regularity also makes decision problems such as policy comparison intractable, in addition to preventing the construction of the view schema. Therefore, we investigate a few restrictions on the views that allow for better algorithms.

Bounded depth A set of trees L has *bounded depth* if there exists a constant k such that all trees in L have depth at most k. In our setting, it is not the depth of the view trees that we wish to bound, but the depth of the original document. Thus, a view $V \subseteq T_{\Sigma\times\Sigma_\varepsilon}$ has bounded depth if there exists some k such that every tree alignment $t \in V$ has depth at most k. We point out that it is not sufficient for the view document to have bounded depth: the whole alignment must have bounded depth: if V has domain D, V has bounded depth if and only if D has bounded depth: this implies that $\mathcal{V}iew\,(V, D)$ has bounded depth, but the latter is not a sufficient condition. For a bounded-depth view, $\mathcal{V}iew\,(V, D)$ is clearly a regular set of trees; this can also be seen as a particular case of Proposition 4.2. Furthermore, Regular XPath and *MSO* clearly have the same expressivity on trees of bounded depth.

Essentially, when the depth is bounded, we can express in Monadic Second Order (over words) the binary relation matching the pairs of corresponding opening and closing tags of the linearization, so that each MSO formula ϕ over trees can be expressed with an "equivalent" MSO formula ϕ_w over (nested) words, such that for every tree t, we have $t \models \phi$ if and only if $lin(t) \models \phi_w$. The axis *child* can be eliminated in ϕ_w, so that ϕ becomes a "standard" boolean MSO formula over words. As such, it defines a regular word language, so the set of words it accepts can be defined with a regular expressions e_w over $\hat{\Sigma}$: for every tree t, we have $t \models \phi$ if and only if $lin(t) \in L(e_w)$. This regular expression over nested words can in turn be simulated by a Regular XPath expression $\mathcal{X}$ over the corresponding trees, so that for every tree t, we have $t \models \mathcal{X}$ if and only if $lin(t) \in L(e_w)$. This shows that for every k, there exists some equivalent Regular XPath expression for every MSO formula over trees of depth bounded by k.

Upward closed views A view is *upward-closed* if it is a set of upward-closed tree alignments. That means all the ancestors of every visible node are also visible. Equivalently, whenever a node is hidden, all its descendants are hidden as well. For this reason, this requirement is commonly referred to in the literature as the policy's *denial downward consistency* [MTKH06] [1].

Interval boundedness We generalize this notion to allow restricted deletions of internal nodes. We say that a tree t over $T_{\Sigma \times \Sigma_\varepsilon}$ is *k-interval bounded* if the following two conditions are satisfied: (1) the label of the root of t belongs to $\Sigma \times \Sigma$ and (2) on any descending path of t, there are at most k consecutive nodes with label in $\Sigma \times \{\varepsilon\}$ between two nodes with label in $\Sigma \times \Sigma$.

A view, or more generally a tree language $L \subseteq T_{\Sigma \times \Sigma_\varepsilon}$ is k-interval bounded if every tree of L is k-interval bounded, and we say that L is *interval bounded* if there exists some k such that L is k-interval bounded. In the same way, we can define k-interval bounded queries and interval bounded queries since every query represents a view. Note that by definition any interval bounded query (or annotation) is obviously root preserving.

Remark 4.1. *Every upward-closed view is* 0*-interval bounded, and every view with bounded depth* k *is* $(k-1)$*-interval bounded.*

We state further properties of interval-bounded *MSO* queries after a few illustrative examples.

[1] The term "upward-closed" is employed by Libkin and Sirangelo [LS10], but a variety of other names appear in the litterature.

Example 4.2. *The security view defined by* (D_0, ann_0) *in Example 4.1 is interval bounded since DTD* D_0 *is non recursive. It is actually (also)* 1-*interval bounded, but not* 0-*interval bounded. The following DTD* D_1 *gives information about the versions of projects.*

$$\begin{array}{l}
\texttt{projects} \to \texttt{project}^* \\
\texttt{project} \to \texttt{name}, \texttt{version} \\
\texttt{version} \to \texttt{number}, \texttt{files}, \texttt{license}, \texttt{prev}
\end{array}$$

$$\begin{array}{l}
\texttt{prev} \to \texttt{version} \mid \varepsilon \\
\texttt{files} \to \texttt{src}, \texttt{bin}, \texttt{doc} \\
\texttt{license} \to \texttt{free} \mid \texttt{propr}
\end{array}$$

Annotation ann_1 *keeps the last version of each project and hides the others. Moreover, it hides all nodes* `version`, `files`, `number` *(when no explicit rule is given for an element name, its visibility is inherited from its parent):*

$$\begin{array}{l}
\texttt{projects} \to \texttt{project}^* \\
\texttt{project} \to \texttt{name}, \texttt{version} \\
\quad \mathsf{ann}_1(\texttt{project}, \texttt{version}) = \mathsf{false} \\
\texttt{version} \to \texttt{number}, \texttt{files}, \texttt{license}, \texttt{prev} \\
\quad \mathsf{ann}_1(\texttt{version}, \texttt{license}) \\
\quad = [\Uparrow::\texttt{version}/\Uparrow::\texttt{project}] \\
\texttt{prev} \to \texttt{version} \mid \varepsilon
\end{array}$$

$$\begin{array}{l}
\texttt{files} \to \texttt{src}, \texttt{bin}, \texttt{doc} \\
\quad \mathsf{ann}_1(\texttt{files}, \texttt{src}) \\
\quad = \mathsf{ann}_1(\texttt{files}, \texttt{bin}) \\
\quad = \mathsf{ann}_1(\texttt{files}, \texttt{doc}) \\
\quad = [\Uparrow::\texttt{files}/\Uparrow::\texttt{version}/\Uparrow::\texttt{project}] \\
\texttt{license} \to \texttt{free} \mid \texttt{propr}
\end{array}$$

The DTD D_1 *is recursive but query* $Q_{(D_1, \mathsf{ann}_1)}$ *is also* 1-*interval bounded, and* $\mathcal{V}iew\big(Q_{(D_1,\mathsf{ann}_1)}, L(D_1)\big)$ *is the language validated by the following DTD* D_1'*:*

$$\begin{array}{l}
\texttt{projects} \to \texttt{project}^* \\
\texttt{project} \to \texttt{name}, \texttt{src}, \texttt{bin}, \texttt{doc}, \texttt{license}
\end{array}$$

$$\begin{array}{l}
\texttt{license} \to \texttt{free} \mid \texttt{propr}
\end{array}$$

The preceding policy is not upward closed as it hides the `version` *nodes that are children of the* `project` *nodes but discloses the* `files` *children of those hidden* `version` *nodes. If we replace, however, the annotation* ann_1 *by* ann_1' *defined by the unique mapping* $\mathsf{ann}_1(\texttt{version}, \texttt{prev}) = \mathsf{false}$*, then the resulting policy is upward-closed (and therefore interval bounded). The corresponding view DTD is* D_1' *given below:*

$$\begin{array}{l}
\texttt{projects} \to \texttt{project}^* \\
\texttt{project} \to \texttt{name}, \texttt{version} \\
\texttt{version} \to \texttt{number}, \texttt{files}, \texttt{license}
\end{array}$$

$$\begin{array}{l}
\texttt{license} \to \texttt{free} \mid \texttt{propr} \\
\texttt{files} \to \texttt{src}, \texttt{bin}, \texttt{doc}
\end{array}$$

Example 4.3. *Let us consider a slightly more complex example: we allow the previous version of project to be a collection of projects. This corresponds to the following case scenario: projects are allowed to merge over time, but not to branch. We define a new DTD* D_2 *obtained from* D_1 *by changing the production of* `prev` *for:* $\texttt{prev} \to \texttt{project}^*$*. All other production rules remain the same.*

Annotation ann_2 *keeps licenses together with the name and version of the corresponding project and the project node, and hides every other node.*

$\texttt{projects} \rightarrow \texttt{project}^*$
$\texttt{project} \rightarrow \texttt{name}, \texttt{version}$
$\quad \mathsf{ann}_2(\texttt{project}, \texttt{name}) = \mathsf{false}$
$\quad \mathsf{ann}_2(\texttt{project}, \texttt{version}) = \mathsf{false}$
$\texttt{version} \rightarrow \texttt{number}, \texttt{files}, \texttt{license}, \texttt{prev}$
$\quad \mathsf{ann}_2(\texttt{version}, \texttt{license}) = \mathsf{true}$
$\texttt{prev} \rightarrow \texttt{project}^*$
$\quad \mathsf{ann}_2(\texttt{prev}, \texttt{project}) = \mathsf{true}$
$\texttt{files} \rightarrow \texttt{src}, \texttt{bin}, \texttt{doc}$
$\texttt{license} \rightarrow \texttt{free} \mid \texttt{propr}$

The query $Q_{(D_2,\mathsf{ann}_2)}$ is not 1-interval bounded, but it is 2-interval bounded. The corresponding view DTD is D_2 given below:

$\texttt{projects} \rightarrow \texttt{project}^*$
$\texttt{project} \rightarrow \texttt{name}, \texttt{license}, \texttt{project}^*$
$\texttt{license} \rightarrow \texttt{free} \mid \texttt{propr}$

As a last example, suppose we only want to store all licenses without further information. This can be achieved, for instance, via annotation ann'_2: $\mathsf{ann}'_2(\texttt{projects}, \texttt{project}) = \mathsf{false}$, and $\mathsf{ann}'_2(\texttt{version}, \texttt{license}) = \mathsf{true}$. The query $Q_{(D_2,\mathsf{ann}'_2)}$ is not interval bounded. The resulting view DTD contains a single production rule: $\texttt{projects} \rightarrow \texttt{license}^*$.

Below we are stating the main property of interval bounded views, namely, that interval bounded views preserve regularity.

Proposition 4.2. *For any regular language D and view V (given as an automaton over tree alignments), if V is interval bounded then the language $\mathcal{V}iew\,(V, D)$ is regular.*

Proof. Let V be a k-interval-bounded view. Let $\mathcal{A} = (\Sigma \times \Sigma_\varepsilon, S_A, \Gamma, I, F, R)$ be a VPA that accepts $\{t \in V \mid \pi_1(t) \in D\}$. We define the VPA $\mathcal{A}'$ as follows: $\mathcal{A}' = (\Sigma, S', \Gamma', I, F, R')$ where $S' = S_A \times \Gamma^{\leqslant k}$, $\Gamma' = \Gamma^{\leqslant k} \times \Gamma$, and the transition rules R' are defined as follows

- $\mathcal{A}'$ has transition $\langle q, w\rangle \xrightarrow{\epsilon} \langle q', w \cdot \gamma\rangle$ for all transition $q \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q'$ in R, and $w \in \Gamma^{<k}$

- $\mathcal{A}'$ has transition $\langle q, w \cdot \gamma\rangle \xrightarrow{\epsilon} \langle q', w\rangle$ for all transition $q \xrightarrow{(\mathrm{cl},(a,\varepsilon)):\gamma} q'$ in R, and $w \in \Gamma^{<k}$

- $\mathcal{A}'$ has transition $\langle q, w\rangle \xrightarrow{(\mathrm{op},b):\langle w,\gamma\rangle} \langle q', \epsilon\rangle$, for all transition $q \xrightarrow{(\mathrm{op},(a,b)):\gamma} q'$ in R, and $w \in \Gamma^{\leqslant k}$.

- $\mathcal{A}'$ has transition $\langle q, \epsilon\rangle \xrightarrow{(\mathrm{cl},b):\langle w,\gamma\rangle} \langle q', w\rangle$ for all transition $q \xrightarrow{(\mathrm{cl},(a,b)):\gamma} q'$ in R, and $w \in \Gamma^{\leqslant k}$.

- $\mathcal{A}'$ has transition $\langle q, w\rangle \xrightarrow{\epsilon} \langle q', w\rangle$ for all $w \in \Gamma^k$ and q, q' such that $\mathcal{A}_{q,q'}$ accepts a tree t over alphabet $\Sigma \times \{\varepsilon\}$.

Let w a word over $\{op, cl\} \times \Sigma$, $\langle q, u\rangle$ a state in S', and σ a word over $\Gamma^{\leqslant k} \times \Gamma$. For the sake of clarity, we denote by σ' the same σ considered as a word over Γ. We claim that for all such w, q, and σ, $\mathcal{A}'$ preserves the following invariant.

> **Invariant:** *$\mathcal{A}'$ can reach configuration $(\langle q, u\rangle, \sigma)$ after reading w if and only if there exists a word w' over $\{op, cl\} \times \Sigma \times \Sigma_\varepsilon$ such that the following two conditions are satisfied: (1) $\pi_2(w') = w$, and (2) $\mathcal{A}$ can reach configuration $(q, \sigma' u)$ after reading w'.*

From this invariant we immediately deduce $L(\mathcal{A}') = \mathcal{V}iew\,(V, D)$. The VPA $\mathcal{A}'$ uses ϵ-transitions to simulate hidden elements. Because of the interval boundedness assumption, the corresponding evolution of the stack can be simulated within the state of the automaton. The last condition corresponds to an ϵ-transition from state q to state q' whenever there is some tree t such that the second component of any label in t is ε and some run of the automaton $\mathcal{A}$ can exit from t in state q' if it enters in state q. Of course those ϵ-transitions can be eliminated. Observe also that existence of the ϵ-transition $\langle q, w\rangle \xrightarrow{\epsilon} \langle q', w\rangle$ does not depend on the value of w. The set of all pairs q and q' satisfying the conditions to obtain an ϵ-transition by the last rule can be computed in time $O(|R|^2 + |S_A|^3)$ or $O(|R| + |S_A|^3 \times |\Gamma|)$ as an adaptation of the algorithm computing $Acc_{\mathcal{A}}$ in Proposition 3.8: we compute the horizontal reachability relation for the VPA obtained from $\mathcal{A}$ by keeping only the transitions with label in $\Sigma \times \{\varepsilon\}$. The whole construction of $\mathcal{A}'$ is therefore polynomial in $\mathcal{A}$ for a fixed value of k, but exponential in k.

Let us now prove the claim. Clearly, $L(\mathcal{A}) \subseteq \pi_2(L(\mathcal{A}')) = \mathcal{V}iew\,(V, D)$. The reverse inclusion also holds due to our interval-boundedness hypothesis. When opening visible elements, $\mathcal{A}'$ records the information of previous simulations in the stack, so that they may be recovered on the corresponding closing tag. This concludes the proof of Proposition 4.2. □

Proposition 4.3. *Let V a view over $\Sigma \times \Sigma_\varepsilon$ given by a VPA $\mathcal{A}$ with N states, then V is interval bounded iff it is $(N^2 + 1)$-IB.*

Proof. We use the vertical pumping argument from Lemma 3.11. Let us suppose that V is k-IB for some k, but not $(N^2 + 1)$-IB. Then there is some tree $t \in V$ such that t is not (N^2+1)-bounded: there is a path in t from some node n to some of its descendants n' such that $lab_t(n')$ and $lab_t(n')$ belong to $\Sigma \times \Sigma$, there are at least $(N^2 + 1)$ nodes on the path between n and n', and all these nodes between n and n' have label in $\Sigma \times \{\varepsilon\}$. Since there are at least $(N^2 + 1)$ such nodes, this implies that on some (in fact, any) accepting run ρ of $\mathcal{A}$ on t, there are two nodes n_1 and n_2 such that $\rho(n_1) = \rho(n_2)$. The vertical pumping argument contradicts the interval-boundedness of Q. □

Proposition 4.4. *For any view V given by an automaton $\mathcal{A}$ over $\Sigma \times \Sigma_\varepsilon$, testing whether V is interval bounded is in* PTime.

Proof (outline). Essentially, the set of all k-IB trees over $\Sigma \times \Sigma_\varepsilon$ can be defined by a deterministic automaton with $O(k)$ states. Hence, it suffices to combine the previous proposition and a simple polynomial algorithm for testing inclusion of tree automata. □

Proposition 4.5. *Testing whether a query given by a $\mathcal{X}Reg$ expression is interval bounded is* EXPTIME*-complete.*

Proof. Building an NTA from a $\mathcal{X}Reg$ expression is in EXPTIME [CGLV09]. Hence, the EXPTIME upper bound follows from Proposition 4.4. To show EXPTIME-hardness, we reduce satisfiability of $\mathcal{X}Reg$ to testing interval boundedness. Let Q a $\mathcal{X}Reg$ expression over an alphabet Σ. We define DTD D as follows: $D = (\Sigma \uplus \{a, b\}, r, P)$ where $P'(r) = \Sigma^* a \mid w \in P(r)\}$, $P(a) = a|b$, $P(b) = \varepsilon$ and, for every $\alpha \in \Sigma \backslash \{r\}$, $P(\alpha) = \Sigma^*$. We rewrite Q in linear time into an expression Q' that checks whether the tree satisfies D and whether Q can be satisfied using only the elements from Σ. If those checks succeed, then Q' selects the (unique) node labeled b, and selects no other node except the root, otherwise it selects only the root. Because the DTD D allows to have b elements at arbitrary depth, the view defined by query Q' is interval bounded iff Q is not satisfiable.

We denote by Q_0 the expression resulting from the addition of a filter $[\,\mathsf{not}(\mathsf{self}::a \text{ } \mathsf{or} \text{ } \mathsf{self}::b)]$ to each elementary axis of Q; for instance every occurrence of $\Rightarrow$ is replaced by $[\,\mathsf{not}(\mathsf{self}::a \text{ } \mathsf{or} \text{ } \mathsf{self}::b)]/\!\Rightarrow\!/[\,\mathsf{not}(\mathsf{self}::a \text{ } \mathsf{or} \text{ } \mathsf{self}::b)]$. We also build in linear time an expression Q_D such that for every tree t, $t \vDash Q_D$ iff $t \in L(D)$. The expression Q' can be built in linear time from Q_D and Q_0:

$$Q' = \Downarrow^*[\,\mathsf{not}\Uparrow \text{ } \mathsf{or} \text{ } (\mathsf{self}::b \text{ } \mathsf{and} \Uparrow^*/[\,\mathsf{not}\Uparrow \text{ } \mathsf{and} \text{ } Q_0 \text{ } \mathsf{and} \text{ } Q_D])] \qquad \square$$

Remark 4.2. *We have shown that interval bounded views preserve regularity. We should note however that they do not preserve $\mathcal{X}Reg$ definability; we show in Proposition 4.37 that any regular tree language L over alphabet Σ is equal to $\mathcal{V}iew\,(Q, T_\Sigma)$ for some 1-interval bounded query $Q \in \mathcal{X}Reg$.*

4.1.3. Evaluation by Query Composition

In the query rewriting approach, the user expresses its queries on the view and those queries must be rewritten into equivalent queries on the original document. Whether such a rewriting process is possible depends on the classes of queries and views involved. We only investigate the case where the query language and the view language have the same expressivity: we prove that this rewriting process is possible for $\mathcal{X}Reg$ queries over $\mathcal{X}Reg$ views, and for *MSO* queries over *MSO* views.

We say that a class $\mathcal{C}$ of queries is *closed under query composition* if for every view $Q_V, Q \in \mathcal{C}$, there exists $Q' \in \mathcal{C}$ such that for all t, $Q'(t) = Q(\mathcal{V}iew\,(Q_V, t))$. This property is called closure under query rewriting in the terminology of [FGJK07], but we avoid this denomination in this dissertation in order to prevent any confusion that may arise with the other notion of "query rewriting" (related to determinacy) that appears in Section 4.2.

Composition of Regular XPath Views The rewriting technique for downward queries from [FCG04] relies on the knowledge of the DTD. Our rewriting method works independently of the DTD. The method uses the fact that visibility of a node can be defined with a single filter expression $\mathcal{X}^{\mathsf{ann}}_{\mathsf{acc}}$ (Lemma 4.1). This filter is used to construct rewritings of the base axes (Lemma 4.6), which are used to rewrite the user queries.

Lemma 4.6. *For any $\mathcal{X}Reg$ query Q and any $\alpha \in \{\Downarrow, \Uparrow, \Rightarrow, \Leftarrow\}$ there exists a $\mathcal{X}Reg$ expression R^Q_α such that $[\![R^Q_\alpha]\!]_t = [\![\alpha]\!]_{\mathcal{V}iew(Q,t)}$ for every tree t. Moreover, $|R^Q_\alpha| = O(|Q|)$.*

Proof. We denote by $Filt(Q)$ the $\mathcal{X}Reg$ filter such that for every tree t and node $n \in N_t$, $t, n \models Filt(Q)$ iff $n \in Q(t)$. Essentially, the rewriting R^Q_α defines paths, traversing inaccessible nodes only, from one accessible node to another accessible node in a manner consistent with the axis α. For the vertical axes the task is quite simple:

$$R^Q_\Downarrow \coloneqq [Filt(Q)]/\Downarrow/([\,\mathsf{not}\,Filt(Q)]/\Downarrow)^*/[Filt(Q)] \quad \text{and} \quad R^Q_\Uparrow \coloneqq (R^Q_\Downarrow)^{-1}$$

Rewritings of the horizontal axes are slightly more complex and we first define auxiliary filter expressions:

$$f^\exists_\downarrow \coloneqq ([\,\mathsf{not}\,Filt(Q)]/\Downarrow)^*/[Filt(Q)], \quad f^\varnothing_\downarrow \coloneqq \mathsf{not}\,f^\exists_\downarrow, \quad f^\varnothing_\rightarrow \coloneqq (\Rightarrow/[f^\varnothing_\downarrow])^*/[\,\mathsf{not}(\Rightarrow)].$$

$f^\exists_\downarrow$ checks that the current node or any of its descendants is accessible. Conversely, $f^\varnothing_\downarrow$ checks whether the current node and all of its descendants are inaccessible. Similarly, $f^\varnothing_\rightarrow$ verifies that only inaccessible nodes can be found among the siblings following the current node and their descendants.

The expression $R^Q_\Rightarrow$ seeks the next accessible node among the following siblings of the current node and their descendants. However, if there are no such nodes but the parent is inaccessible, the next accessible node is sought among the following siblings of the parent. The last step is repeated recursively if needed.

$$R^Q_\Rightarrow \coloneqq [Filt(Q)]/([f^\varnothing_\rightarrow]/\Uparrow/[\,\mathsf{not}\,Filt(Q)])^*/\Rightarrow/(\phi_1 \cup \phi_2)^*/[Filt(Q)] \text{ and } R^Q_\Leftarrow \coloneqq (R^Q_\Rightarrow)^{-1}$$

where $\phi_1 = [(\,\mathsf{not}\,Filt(Q))\ \mathsf{and}\ f^\varnothing_\downarrow]/\Rightarrow$ and $\phi_2 = [(\,\mathsf{not}\,Filt(Q))\ \mathsf{and}\ f^\exists_\downarrow]/\Downarrow/[\neg\Leftarrow]$. We observe that $|R^Q_\alpha| = O(|Q|)$ for every $\alpha \in \{\Downarrow, \Uparrow, \Rightarrow, \Leftarrow\}$. □

Theorem 4.7. *$\mathcal{X}$Reg is closed under query composition. Moreover, given a $\mathcal{X}$Reg query Q and a root-preserving $\mathcal{X}$Reg query Q', we can compute a $\mathcal{X}$Reg formula* $\mathsf{Rewrite}(Q, Q')$ *in time $O(|Q| * |Q'|)$ such that, for every tree t,* $\mathsf{Rewrite}(Q, Q')(t) = Q(\mathcal{V}iew\,(Q', t))$.

Proof. The formula $\mathsf{Rewrite}(Q, Q')$ replaces in Q every occurrence of a base axis $\alpha \in \{\Downarrow, \Uparrow, \Rightarrow, \Leftarrow\}$ with $R_\alpha^{Q'}$. A simple induction over the size of Q shows that $[\![Q]\!]_{\mathcal{V}iew(Q',t)} = [\![\mathsf{Rewrite}(Q, Q')]\!]_t$, Lemma 4.6 handling the nontrivial base cases. Since the root is always accessible, we get $Q\,(\mathcal{V}iew\,(Q', t)) = \mathsf{Rewrite}(Q, Q')\,(t)$. We note that the rewritten query is constructed in time $O(|Q| * |Q'|)$, which also bounds the size of $\mathsf{Rewrite}(Q, Q')$. □

We observe that the asymptotic complexity of our rewriting method is comparable to that of [FGJK06] but it handles a larger class of queries and DTDs.

Our result is quite similar to the corresponding results of closure under composition for other fragments of XPath in [VHP06]. In both algorithms – ours and theirs – the composition of the view and query are essentially obtained by rewriting the base axes. On the one hand we handle a more expressive fragment of XPath, but on the other hand some queries can be expressed more succinctly using XPath 2.0's path complementation and intersection than with the Kleene star of Regular XPath. While the rewriting of the base axes in [VHP06] relies on those path complementation and intersection operators, our rewriting only uses the simpler Boolean negation and conjunction of filters. To be fair, let us observe that the crux of our rewriting algorithm is already present in [VHP06].

Composition of Views Defined by Automata MSO enjoys the same closure under composition as Regular XPath: given two query automata (or even view automata) Q and Q_v, we can compute a query automaton (resp. view automaton) in polynomial time for the composition of Q and Q_v. The rewriting is obtained through a standard construction by synchronization of Q and Q_v, resulting in a new query automaton $\mathsf{Rewrite}(Q, Q_v)$.

Theorem 4.8. *Query automata are closed under query composition, i.e., for every root-preserving query automaton Q_v, and every query automaton Q, there exists a query automaton* $\mathsf{Rewrite}(Q, Q_v)$ *such that* $Q\,(\mathcal{V}iew\,(Q_v$
$\mathsf{Rewrite}(Q, Q_v)\,(t)$. *Moreover, we can compute* $\mathsf{Rewrite}(Q, Q_v)$ *in time $|Q| \times |Q_v|$. View automata can be composed likewise.*

Proof. From the two automata $Q_v = (\Sigma \times \Sigma_\varepsilon, S_v, \Gamma_v, I_v, F_v, R_v)$ and $Q = (\Sigma \times \Sigma_\varepsilon, S, \Gamma, I, F, R)$, we build automaton $Q_\circ = (\Sigma \times \Sigma_\varepsilon, S_\circ, \Gamma_\circ, I_\circ, F_\circ, R_\circ) =$

$\mathsf{Rewrite}(Q, Q_v)$ as follows: $S_\circ = S_v \times S$, $\Gamma_\circ = \Gamma_v \times (\Gamma \cup \{\#\})$, $I_\circ = I_v \times I$, $F_\circ = F_v \times F$, and the transitions are defined by the two following rules: (1) we add transition $(s_v, s) \xrightarrow{(\eta,(a,\varepsilon)):(\gamma_v,\#)} (s'_v, s)$ to $R_\circ$, for every $\eta \in \{op, cl\}$, $s_v, s'_v \in S_v$, $a \in \Sigma$, $\gamma_v \in \Gamma_v$, every transition $s_v \xrightarrow{(\eta,(a,\varepsilon)):\gamma_v} s'_v \in R_v$, and every $s \in S$, and (2) we add transition $(s_v, s) \xrightarrow{(\eta,(a,\beta)):(\gamma_v,\gamma)} (s'_v, s')$ to $R_\circ$ for every $\eta \in \{op, cl\}$, $s_v, s'_v \in S_v$, $a \in \Sigma$, $\gamma_v \in \Gamma_v$, $s, s' \in S, \gamma \in \Gamma, \beta \in \Sigma_\varepsilon$, every transitions $s_v \xrightarrow{(\eta,(a,a)):\gamma_v} s'_v \in R_v$ and $s \xrightarrow{(\eta,(a,\beta)):\gamma} s' \in R$. The number of transitions added by the first rule is at most $|R_v| \times |S|$, whereas the number of transitions added by the second rule is at most $|R_v| \times |R|$, which sums up to $O(|Q_v| \times |Q|)$. The resulting automaton $Q_\circ$ satisfies $Q_\circ(t) = Q(\mathcal{V}iew(Q_v, t))$ for every tree t, since it satisfies the following invariant.

> **Invariant:** *For every word w over $\{op, cl\} \times \Sigma \times \Sigma_\varepsilon$ and every state $(s_v, s) \in S_\circ$, there exists some word u over $\Gamma_\circ$ such that $\mathcal{A}_\circ$ reaches $((s, s_v), u)$ after reading w if and only if there exist a word w' over $\{op, cl\} \times (\Sigma \times \Sigma_\varepsilon \cup \Sigma \times \{\varepsilon\}^2)$ and two words u_1 and u_2 over Γ_v and Γ such that the following three conditions are satisfied:*
>
> 1. $\pi_{1,3}(w') = w$
>
> 2. *Q_v reaches (s_v, u_1) after reading $\pi_{1,2}(w')$, and*
>
> 3. *Q reaches (s, u_2) after reading $\pi_{2,3}(w')$.* □

In a nutshell, we have proved that $\mathcal{X}Reg$ and *MSO* are closed under query composition, and that we can compute a rewriting in polynomial time. The same proof shows the closure under composition of view automata: given any view automata A_v and A, we can compute a view automaton $\mathsf{Rewrite}(A, A_v)$ such that for all t, $\mathcal{V}iew(A, \mathcal{V}iew(A_v, t)) = \mathcal{V}iew(\mathsf{Rewrite}(A, A_v), t)$: the only difference being the condition for adding transition $(s_v, s) \xrightarrow{(\eta,(a,\beta)):(\gamma_v,\gamma)} (s'_v, s')$ to $R_\circ$ becoming the existence of $\eta \in \{op, cl\}$, $s_v, s'_v \in S_v$, $a, b \in \Sigma$, $\gamma_v \in \Gamma_v$, $s, s' \in S, \gamma \in \Gamma$, and $\beta \in \Sigma_\varepsilon$, such that $s_v \xrightarrow{(\eta,(a,b)):\gamma_v} s'_v$ belongs to R_v and $s \xrightarrow{(\eta,(b,\beta)):\gamma} s'$ to R.

Materialized vs. Non-materialized Views We review the worst-case performance of querying in the materialized and non-materialized setting in a scenario with a single query, view and without updates to the document. We then discuss the relevance and limits of such a comparison. We recall that we compare the following two methods for evaluating of two different methods for evaluating a query over the view: either the view is first materialized and then queried, or the query is first rewritten to include the view query and directly evaluated on original document.

Let t a document, Q_v a query defining the view and Q' a query. When Q_v and Q' are $\mathcal{X}Reg$ queries, the cost of materialization would be $\Theta(|Q_v| \times |t|)$, so that the overall cost of evaluating Q' on t is $\Theta(|Q'| \times |\mathcal{V}iew\,(Q_v, t)\,|) + \Theta(|Q_v| \times |t|)$ which amounts to $\Theta((|Q'| + |Q_v|) \times |t|)$ in the worst case. In the non-materialized setting, the complexity of evaluating $\mathsf{Rewrite}(Q', Q_v)$ over t could appear to be $\Theta(|\mathsf{Rewrite}(Q', Q_v)| \times |t|)$ if we evaluate naively $\mathsf{Rewrite}(Q', Q_v)$, however the multiple occurrences of $\mathit{Filt}(Q)$ do not raise the complexity w.r.t. a single occurrence, so that the cost of evaluating $\mathsf{Rewrite}(Q', Q_v)$ can be reduced to $\Theta((|Q'| + |Q_v|) \times |t|)$. This can be established either by analysing the dynamic programming algorithm for evaluating PDL [AI00, Mar04], or by observing that in the 2-ATA one can build from $\mathsf{Rewrite}(Q', Q_v)$, the multiple occurrences of $\mathcal{X}^{Q_v}_{\text{acc}}$ can be represented with the same state: thus the conversion of $\mathsf{Rewrite}(Q', Q_v)$ into a 2-ATA is in $\Theta(|Q_v| + |Q'|)$, so that we can evaluate this 2-ATA over t in time $\Theta((|Q_v| + |Q'|) \times |t|)$ by [CGLV09, KVW00]. Consequently, when we compare the worst-case complexity of query evaluation, the non-materialized setting does not improve upon the view materialization, but it does not make things worse either (no wonder, as a matter of fact, since the evaluation of $\mathcal{X}\mathit{Reg}$ in quadratic time actually simulates the materialization).

When Q_v and Q' are VPAs, the cost of materialization rises to roughly $\Theta(|Q_v|^3 \times |t|)$ (see Proposition 3.9), while querying the materialized views induces an additional $\Theta(|Q'|^3 \times |\mathcal{V}iew\,(Q_v, t)\,|)$, which matches $\Theta(|Q'|^3 \times |t|)$ in the worst case. The overall complexity of query evaluation in the materialized setting sums up to $\Theta((|Q_v| + |Q'|)^3 \times |t|)$. In the non-materialized setting, computing the automaton for $\mathsf{Rewrite}(Q', Q_v)$ requires $\Theta(|Q_v| \times |Q'|)$. Therefore, the overall cost of query evaluation is raised to $\Theta(|Q_v|^3 \times |Q'|^3 \times |t|)$ if we do not consider potential optimizations.

Those worst-case bounds are not very relevant for general scenarios, however. First, the worst case may seem unlikely and small view trees would favor the materialized setting. Moreover, in the materialized framework, the view document is computed once and for all, as long as no update is applied to the document: once the view has been materialized multiple queries can be processed on the view, which allows to amortize the cost of materialization.

The main assets of non-materialized views are gains in terms of space required for storage, and avoiding the need to compute the view(s) anew after each update of the document. The non-materialized setting may thus be more interesting only when the number of roles having different privileges and the frequency of updates are not dwarfed by the number of queries. Even in that case, the view may be evaluated/updated “on-demand” at query time, in which case the comparison scenario above makes sense, and tends to question the point of non-materialized views in terms of time efficiency.

On the other hand, the scenario does not take into account optimization techniques, which may favor either of the methods. Techniques such as incremental evaluation allow to cope with updates in the materialized framework, while pruning techniques may enhance the effectiveness of the non-materialized setting.

4.1.4. Annotated DTD Models for Query Rewriting

The specification of access control policies by annotating pairs of labels as above is common to the models of Fan et al. [FCG04, FGJK07] and Kuper et al. [Ras07, KMR09], yet the view derived from the same specification differs.

The View Specification In the models of Fan et al., and Kuper et al, the annotated DTD (D, ann)[2] is used to derive a *security view*. In the terminology of [KMR05, KMR09, FCG04, FGJK07], a security view is a pair (D_v, σ), with D_v the view DTD, and σ a function that maps each edge of D_v, i.e., each pair of labels A, B such that B occurs in the production of A in D_v, to an XPath query $\sigma(A, B)$. Given an accessible A-labeled node n in some document t, evaluating the query $\sigma(A, B)$ at node n allows to extract the B node(s) below A whose parent in the materialized view is node n. Thus, $\sigma(A, B)$ returns the B labeled nodes that are "directly" accessible from n.

In those models, the annotation is defined as a function ann mapping pairs of symbols to Y, N or an XPath filter. Y stands for Yes (true in our model) and N for No (false in our model). Unlike Kuper et al. [KMR05], Fan et al. [FCG04] only consider normalized DTDs. In a normalized DTD the production rules are of the form: $a \to \varepsilon$, $a \to b^*$, $a \to b_1, b_2, \dots, b_n$, or $a \to b_1 \mid b_2 \mid \dots \mid b_n$ (with pairwise distinct elements b_i). In particular, each rule assigns to the nodes a bounded number of distinct elements except for the rule $a \to b^*$. Both the original DTD and the view DTD must be normalized DTDs.

Different Semantics for Annotated DTDs In [FCG04], inaccessible nodes are generally anonymized instead of being deleted, essentially to guarantee that the view DTD is normalized. For instance, consider a DTD with production rules $a \to b_1 + b_2 + \cdots + b_n$ and $b_2 \to c_1 + \cdots + c_n$ with annotation $\mathsf{ann}(a, b_2) = \mathsf{false}$ and $\mathsf{ann}(b_2, c_i) = \mathsf{true}$ for all i. Then b_2 is deleted from the view and the view DTD has production rules: $a \to b_1 + c_1 + \dots c_n + \cdots + b_n$. If however the production rule for b_2 is $b_2 \to c_1, \dots, c_n$, then b_2 is anonymized into a dummy label x, and the view DTD is defined by the rules $a \to b_1 + x + \cdots + b_n$ and $x \to c_1, \dots, c_n$. This normalization of DTDs is not so restrictive as every DTD can be turned into a normalized DTD through the insertion of new labels. Using such normalized DTDs simplifies the query rewriting algorithm, but essentially prevents the administrator from hiding information on the structure of the document.

The choice of Kuper et al. is to delete systematically the invisible nodes. This is also the semantics we adopted for annotated DTDs, although view automata can also specify relabelings in addition to deletions.

[2] a.k.a. *access specification* [FCG04] or *authorization specification* [Ras07, KMR09]

Another specificity of annotations in [FCG04] is the semantics of filters: Given a b-labeled node n with parent labeled a, if $\mathsf{ann}(a, b)$ is a filter $[q]$ that evaluates to false at n, then the whole subtree below b is deleted, whereas for an annotation $\mathsf{ann}(a, b) = N$, visible descendants of node n are present in the view.

Query Rewriting Algorithms As already mentioned in Chapter 2, the fragments of XPath used in the annotation vary: XPath$(\Downarrow, \Downarrow^*, \cup, [\ ], \wedge, \vee, \neg)$ for [FCG04], XPath$(\Downarrow, \Downarrow^*, \Uparrow, \Uparrow^*, \cup, [\ ], \wedge, \vee, \neg)$ for [Ras07, KMR09], and Regular XPath$(\Downarrow, \Downarrow^*, \cup, [\,], \wedge, \vee, \neg)$ for [FGJK07].

The query rewriting algorithms of those models [FCG04, Ras07, FGJK07] essentially rely on σ to compute the composition of the view and query. The algorithms from [FCG04] and [Ras07] are very close to each other, but the algorithm from [FGJK07] is more distantly related to the others due to the choice of *mixed finite state automata* as a query model. The authors optimize the complexity for query evaluation with a specific evaluation algorithm for their mixed finite state automata: they can evaluate an automaton $\mathcal{A}$ over a document t in time $O(|\mathcal{A}| \times |t|)$. The complexity of evaluating a Regular XPath query over the security view (D_v, σ) for any document t amounts to $O(|Q|^2|\sigma||D_V|^2 + |Q||\sigma||D_V||t|)$ in the model of Fan et al. for recursive views and queries [FGJK07]: the automaton for the composition of the view and query is obtained in time $O(|Q|^2|\sigma||D_V|^2)$, and has size $O(|Q||\sigma||D_V|)$, which yields the above complexity.

4.2. Comparing Policies

4.2.1. How can we Compare Policies?

We wish to provide the administrator with tools for comparing access control policies. Although the access control policy is not disclosed in our model, we implicitly suppose in this section that an "attacker" may obtain full knowledge of the access control policy, and the information we are protecting is the source document. We first address this problem when the views do not relabel nodes, and then discuss how the results can be extended to views that are allowed to relabel nodes.

Inclusion of Queries

A straightforward approach is to compare the nodes made visible by the root preserving queries:

Definition 4.1 (inclusion). *Given two root preserving queries Q_1 and Q_2 with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, we say that Q_1 is included in Q_2 and write $Q_1 \preccurlyeq_1 Q_2$ if $Q_1(t)$ is a subset of $Q_2(t)$ for every t in D.*

The first comparison thus establishes whether all nodes visible for Q_1 are also displayed by query Q_2. This comparison may sometimes be deficient: when $Q_1 \preccurlyeq_1 Q_2$, Q_1 may still disclose some information that Q_2 does not, while hiding more nodes than Q_2, because a malicious user that knows some information on the access control policy might infer information about the origin of a node, as illustrated in the following example.

Example 4.4. *We consider the DTD D_0 given in Example 4.1, with another annotation ann'_0. In this annotation, nodes* `src` *under* `dev` *are always hidden (not only when they are under a proprietary licensed project). So the last rule of ann_0 is replaced by:*

$$\begin{aligned}&\mathtt{dev} \to \mathtt{src}, \mathtt{doc}\\ &\quad \mathsf{ann}'_0(\mathtt{dev}, \mathtt{src}) = \mathsf{false}\\ &\quad \mathsf{ann}'_0(\mathtt{dev}, \mathtt{doc}) = \mathsf{true}\end{aligned}$$

In this example, annotation ann'_0 hides more nodes than ann_0, so $Q_{(D_0,\mathsf{ann}'_0)} \preccurlyeq_1 Q_{(D_0,\mathsf{ann}_0)}$, as evidenced by Figures 4.1 and 4.2. But hiding nodes may reveal some information. Indeed, for every t valid for the DTD D_0, the projects with free license that are currently under development can be selected with the following $\mathcal{X}Reg$ expression on $\mathcal{V}iew\left(Q_{D_0,\mathsf{ann}'_0}, t\right)$:

$$\Downarrow::\mathtt{projects}/\Downarrow::\mathtt{project}[\mathtt{not}(\Downarrow::\mathtt{src})\ \mathtt{and}\ \Downarrow::\mathtt{license}/\Downarrow::\mathtt{free}]$$

So the user can distinguish some projects under development from stable projects, which was not possible with ann_0.

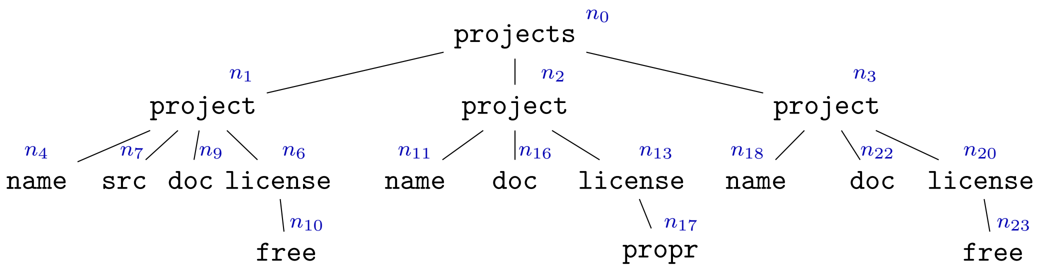


Figure 4.2.: The view $\mathcal{V}iew\left(Q_{D_0,\mathsf{ann}'_0}, t_0\right)$.

For this reason, we define now another way to compare views (root preserving queries), based on which queries can be expressed through the view.

Comparison $\preccurlyeq_{2,\mathcal{C}}$ and Expressible Queries

Given a class of queries $\mathcal{C}$ and a root preserving query Q in $\mathcal{C}$, we define the class of "expressible queries" over the source document as

$$Public^{\mathcal{C}}(Q) = \{Q' \in \mathcal{C} \mid \exists Q'' \in \mathcal{C}.\forall t \in \mathrm{dom}(Q).Q''(\mathcal{V}iew\,(Q, t)) = Q'(t)\}.$$

Definition 4.2. *Given a class of queries $\mathcal{C}$ and two queries Q_1 and Q_2, we write $Q_1 \preccurlyeq_{2,\mathcal{C}} Q_2$ if $\mathit{Public}^{\mathcal{C}}(Q_1) \subseteq \mathit{Public}^{\mathcal{C}}(Q_2)$.*

We call a class of queries $\mathcal{C}$ *well-behaved* if it satisfies the two following conditions: (1) query Q_{all} belongs to $\mathcal{C}$, where for every tree t, $Q_{\text{all}}(t)$ selects all the nodes of t, and (2) $\mathcal{C}$ is closed under query rewriting. Regular XPath and *MSO* are well-behaved classes of queries.

Proposition 4.9. *Given a well-behaved $\mathcal{C}$, and two root preserving queries Q_1 and Q_2 in $\mathcal{C}$ with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$, $Q_1 \preccurlyeq_{2,\mathcal{C}} Q_2$ if and only if there exists some query $Q \in \mathcal{C}$ such that for every t, $Q(\mathcal{V}\mathit{iew}\,(Q_2, t)) = Q_1(t)$, i.e., if and only if $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_2)$.*

Proof. Suppose $\mathit{Public}^{\mathcal{C}}(Q_1) \subseteq \mathit{Public}^{\mathcal{C}}(Q_2)$. Since we supposed Q_{all} belongs to $\mathcal{C}$, $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_1)$, so $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_2)$. Conversely suppose $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_2)$, and let Q denote some query in $\mathcal{C}$ such that for all t in D,

$$Q(\mathcal{V}\mathit{iew}\,(Q_2, t)) = Q_1(t). \tag{4.1}$$

Observe that, since Q_1 is root preserving, so is Q. Fix also $Q' \in \mathit{Public}^{\mathcal{C}}(Q_1)$ and let Q'' denote some query such that for all t in D,

$$Q''(\mathcal{V}\mathit{iew}\,(Q_1, t)) = Q'(t). \tag{4.2}$$

As we supposed $\mathcal{C}$ to be closed under query rewriting, there exists a query Q_r in $\mathcal{C}$ such that for all t,

$$Q_r(t) = Q''(\mathcal{V}\mathit{iew}\,(Q, t)). \tag{4.3}$$

Combining equations (4.1), (4.2) and (4.3) we obtain that for all t in D, $Q_r(\mathcal{V}\mathit{iew}\,(Q_2, t)) = Q'(t)$, hence $Q' \in \mathit{Public}^{\mathcal{C}}(Q_2)$, which concludes our proof. □

To sum up, $Q_1 \preccurlyeq_{2,\mathcal{C}} Q_2$ essentially means that every information we could retrieve from Q_1 using some query from class $\mathcal{C}$ could also be retrieved from Q_2 using some query from class $\mathcal{C}$. For well-behaved classes of queries $\mathcal{C}$, this amounts to deciding whether the first view can be expressed with a query Q'' from $\mathcal{C}$ over the second view, a classical problem of database theory known as *query rewriting.*

With a very large class of queries $\mathcal{C}$, Q'' might prove exceedingly expensive to evaluate, in which case Q_1 would arguably provide some information unavailable (or difficult to obtain) from Q_2. The weaker the class of queries $\mathcal{C}$, the safer Q_1 will be compared to Q_2, but also the more restrictive the comparison.

In the extreme case we may decide not to consider the difficulty of evaluating Q''; for instance if we assume an “adversary” with unlimited computational power. Thus, when $\mathcal{C}$ is the class of all queries (all queries closed under isomorphism), the question is only about whether Q_1 discloses any information hidden by Q_2. We prove below that this case corresponds to the usual notion of *determinacy* for views.

Determinacy, the Least Restrictive Form of Comparison $\preccurlyeq_{2,\mathcal{C}}$

Q_1 will be more secure than Q_2 for comparison $\preccurlyeq_2$ if view Q_2 determines view Q_1, i.e., if we can simulate view Q_1 from view Q_2.

Definition 4.3 (determinacy). *Given two root preserving queries Q_1 and Q_2 with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$*, we say that Q_2 determines Q_1 and write $Q_1 \preccurlyeq_2 Q_2$ if $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t') \implies \mathcal{V}iew\,(Q_1, t) = \mathcal{V}iew\,(Q_1, t')$ for every t and t' in D.*

We also observe that the notion of determinacy comes in two different flavors depending on whether we reason up to isomorphism or prefer to take identifiers into account. If we wish to reason only up to isomorphism of the tree, then the identifiers do not matter and the comparison above can be adapted into $\preccurlyeq_3$ as follows:

Definition 4.4 (determinacy modulo isomorphism). *Given two root preserving queries Q_1 and Q_2 with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$*, we say that Q_2 determines Q_1 modulo isomorphism and write $Q_1 \preccurlyeq_3 Q_2$ if*

$$\forall t, t' \in D . \mathcal{V}iew\,(Q_2, t) \simeq \mathcal{V}iew\,(Q_2, t') \implies \mathcal{V}iew\,(Q_1, t) \simeq \mathcal{V}iew\,(Q_1, t')$$

As we only consider queries closed under isomorphism, $Q_1 \preccurlyeq_2 Q_2$ implies $Q_1 \preccurlyeq_1 Q_2$.

Proposition 4.10. *For any two root preserving queries Q_1 and Q_2 with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$*, $Q_1 \preccurlyeq_2 Q_2$ implies $Q_1 \preccurlyeq_1 Q_2$.*

Proof. Let Q_1, Q_2 two queries with domain D such that $Q_1 \preccurlyeq_2 Q_2$. Suppose $Q_1(t) \nsubseteq Q_2(t)$. There exists some t and a node n in $Q_1(t)$ such that $n \notin Q_2(t)$. Let t' be the tree obtained from t by replacing n with a "fresh" node $n' \notin N_t$ (modifying the relations *child*, *follow* accordingly). As Q_2 and Q_1 are closed under isomorphism, $\mathcal{V}iew\,(Q_2, t') = \mathcal{V}iew\,(Q_2, t)$, and $Q_1(t') \neq Q_1(t)$. This contradicts our hypothesis. Therefore, we must have $Q_1(t) \subseteq Q_2(t)$. □

We only used the fact that for any trees t, t', $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ implies $Q_1(t) = Q_1(t')$ to deduce that $Q_1 \preccurlyeq_1 Q_2$. As a result, we can give a weaker but equivalent formulation for determinacy:

Remark 4.3. *Given two root preserving queries Q_1 and Q_2 with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$*, Q_2 determines Q_1 if and only if $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ implies $Q_1(t) = Q_1(t')$ for every t and t' in D.*

The direct implication is straightforward, and the only-if direction follows from the argument above: If $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t') \implies Q_1(t) = Q_1(t')$ for every t and t' in D, then $Q_1 \preccurlyeq_1 Q_2$, so that from $Q_1(t) = Q_1(t')$ and $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ we deduce $\mathcal{V}iew\,(Q_1, t) = \mathcal{V}iew\,(Q_1, ')$.

We claimed before that comparison $\preccurlyeq_{2,\mathcal{C}}$ becomes comparison $\preccurlyeq_2$ when $\mathcal{C}$ is the class of all queries closed under isomorphism. Let us establish this claim.

Remark 4.4. *Comparison $\preccurlyeq_2$ corresponds to $Q_1 \preccurlyeq_{2,\mathcal{C}} Q_2$ with $\mathcal{C}$ the set of all queries closed under isomorphism.*

Assume first that $Q_1 \preccurlyeq_2 Q_2$, and let $\mathcal{C}$ the class of all queries closed under isomorphism. Let Q' any query in $\mathit{Public}^{\mathcal{C}}(Q_1)$ and Q'' a query such that for all $t \in \mathrm{dom}(Q_1)$, $Q''(\mathcal{V}\mathit{iew}\,(Q_1, t)) = Q'(t)$. For each tree $t_2 \in \mathcal{V}\mathit{iew}\,(Q_2, D)$, every tree t such that $\mathcal{V}\mathit{iew}\,(Q_2, t) = t_2$ returns the same view by Q_1, since $Q_1 \preccurlyeq_2 Q_2$. Therefore, the value of $Q'(t)$ is the same for every such tree. Recall that in this case $Q_1 \preccurlyeq_1 Q_2$ according to Proposition 4.10, so that $Q'(t) \subseteq Q_1(t) \subseteq Q_2(t) = N_{t_2}$. Thus, $Q'(t)$ is a subset of $Q_2(t)$ and only depends on $\mathcal{V}\mathit{iew}\,(Q_2, t)$. We can therefore define a query Q_r that "computes" $Q'(t)$ from $\mathcal{V}\mathit{iew}\,(Q_2, t)$ (the nodes selected by Q_r on tree t_2 are obtained by choosing arbitrarily a tree t such that $\mathcal{V}\mathit{iew}\,(Q_2, t) = t_2$, and then selecting the nodes in $Q'(t)$). The resulting query Q_r is closed under isomorphism because Q' and Q_2 are so. Thus $Q' \in \mathit{Public}^{\mathcal{C}}(Q_2)$, and more generally $\mathit{Public}^{\mathcal{C}}(Q_1) \subseteq \mathit{Public}^{\mathcal{C}}(Q_2)$. Conversely, assume $\mathit{Public}^{\mathcal{C}}(Q_1) \subseteq \mathit{Public}^{\mathcal{C}}(Q_2)$ for some $\mathcal{C}$ that contains Q_1, which is the case for the class we are discussing. In particular, $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_2)$, hence $Q_1 \in \mathit{Public}^{\mathcal{C}}(Q_2)$, which implies $Q_1 \preccurlyeq_2 Q_2$. This concludes the proof.

Remark 4.5. *If we consider views as tree transformations, then both comparisons $\preccurlyeq_2$ and $\preccurlyeq_3$ can be interpreted as a problem of functionality. Intuitively, given views Q_1 and Q_2, one wishes to test if the transformation $t \mapsto Q_1(Q_2^{-1}(t))$ is functional, i.e., maps t to a unique tree (modulo isomorphism for comparison* 3*). In general, however, we shall prove that this transformation can be expressed neither as a (regular) set of 2-alignments, nor as any other representation of which functionality is decidable.*

Comparison $\preccurlyeq_3$ and Certain Answers

Definition 4.5. *Given a root preserving query Q_v, a Boolean query $\mathcal{Q}$, and a tree t_v in $\mathcal{V}\mathit{iew}\,(Q_v, \mathrm{dom}(Q_v))$, we define the set of* possible source documents *of t_v for Q_v as* $\mathrm{Src}\,(t_v, Q_v) = \{t \in \mathrm{dom}(Q_v) \mid \mathcal{V}\mathit{iew}\,(Q_v, t) \simeq t_v\}$. *The* certain answer *of query $\mathcal{Q}$ for t_v is*

$$\mathit{Certain}_{Q_v}\,(\mathcal{Q}; t_v) = \bigwedge_{t \in \mathrm{Src}(t_v, Q_v)} \mathcal{Q}(t).$$

We introduce conditions on the class of queries considered in order to obtain for comparison $\preccurlyeq_3$ an alternative characterization similar to the one obtained for comparison $\preccurlyeq_{2,\mathcal{C}}$.

Definition 4.6. *We say that a class of queries* permits view-inversion *if for every root preserving query $Q_1 \in \mathcal{C}$, any tree $t' \in \mathcal{V}\mathit{iew}\,(Q_1, \mathrm{dom}(Q_1))$, there is a Boolean query* $\mathrm{Ant}\,(t', Q_1)$ *in $\mathcal{C}$ such that $\forall t \in \mathrm{dom}(Q_1).\mathrm{Ant}\,(t', Q_1)\,(t) =$* true *iff $t \in \mathrm{Src}\,(t', Q_1)$, i.e., iff $\mathcal{V}\mathit{iew}\,(Q_1, t) \simeq t'$.*

This means query $\mathrm{Ant}(t', Q_1)$ is only satisfied on trees whose view (for Q_1) is isomorphic to t'.

Lemma 4.11. *Regular XPath and MSO permit view-inversion.*

Proof. Let Q_1 denote a root-preserving $\mathcal{X}Reg$ query and let t' denote a tree in $\mathcal{V}iew(Q_1, \mathrm{dom}(Q_1))$. We can easily define a boolean query $f \in \mathcal{X}Reg$ such that for every tree t, $f \models t$ if and only if $t \simeq t'$. The construction for the composition of queries in Section 4.1 can be applied to boolean queries as well as root-preserving queries; thus, by rewriting the base axes of f, we obtain a $\mathcal{X}Reg$ query $\mathsf{Rewrite}(f, Q_1)$ which for every tree t satisfies $\mathsf{Rewrite}(f, Q_1)(t) = \mathsf{true}$ if and only if $\mathcal{V}iew(Q_1, t) \simeq t'$. The proof for *MSO* follows the same lines. □

For a class $\mathcal{C}$ that permits view inversion and with queries closed under isomorphism, $Q_1 \preccurlyeq_3 Q_2$ iff the certain answers for t with view Q_1 are also certain answers with view Q_2 for every query of $\mathcal{C}$:

Proposition 4.12. *Let $\mathcal{C}$ a class permitting view inversion, and $Q_1, Q_2 \in \mathcal{C}$. Then $Q_1 \preccurlyeq_3 Q_2$ if and only if*

$$\forall t \in D. \forall \mathcal{Q} \in \mathcal{C}. Certain_{Q_1}(\mathcal{Q}; \mathcal{V}iew(Q_1, t)) \implies Certain_{Q_2}(\mathcal{Q}; \mathcal{V}iew(Q_2, t))$$

Proof. Suppose that for all t in D and all $\mathcal{Q}$ in $\mathcal{C}$, $Certain_{Q_1}(\mathcal{Q}; \mathcal{V}iew(Q_1, t))$ implies $Certain_{Q_2}(\mathcal{Q}; \mathcal{V}iew(Q_2, t))$. Since $\mathcal{C}$ permits view inversion, for all t in D, $\mathrm{Ant}(t, Q_1)$ can be expressed with a query in $\mathcal{C}$ and therefore we have $Certain_{Q_1}(\mathrm{Ant}(\mathcal{V}iew(Q_1, t), Q_1); \mathcal{V}iew(Q_1, t)) = \mathsf{true}$. By hypothesis, this implies $Certain_{Q_2}(\mathrm{Ant}(\mathcal{V}iew(Q_1, t), Q_1); \mathcal{V}iew(Q_2, t))$.

Suppose now that $Certain_{Q_2}(\mathrm{Ant}(\mathcal{V}iew(Q_1, t), Q_1); \mathcal{V}iew(Q_2, t)) = \mathsf{true}$ for all t in D, and fix some t, t' in D such that $\mathcal{V}iew(Q_2, t) \simeq \mathcal{V}iew(Q_2, t')$. Then, $\mathrm{Ant}(\mathcal{V}iew(Q_1, t), Q_1)(t') = \mathsf{true}$, hence $\mathcal{V}iew(Q_1, t) \simeq \mathcal{V}iew(Q_1, t')$.

To conclude, suppose that for all t and t' in D, $\mathcal{V}iew(Q_2, t) \simeq \mathcal{V}iew(Q_2, t')$ implies $\mathcal{V}iew(Q_1, t) \simeq \mathcal{V}iew(Q_1, t')$. Then, $\mathrm{Src}(\mathcal{V}iew(Q_2, t), Q_2)$ is a subset of $\mathrm{Src}(\mathcal{V}iew(Q_1, t), Q_1)$ for every t in D. Hence for every t in D and Q in $\mathcal{C}$, $Certain_{Q_1}(\mathcal{Q}; \mathcal{V}iew(Q_1, t)) \implies Certain_{Q_2}(\mathcal{Q}; \mathcal{V}iew(Q_2, t))$. □

Thus, all our comparisons turn out to correspond to standard definitions from database theory; inclusion, determinacy and query rewriting, as mentioned in chapter 2. We henceforth use these characterizations to investigate the decidability of the comparisons.

4.2.2. Preliminary Results Relating the Different Comparisons

The following results describe how the three definitions for policy comparison are related (for queries closed under isomorphism):

Proposition 4.13. *Given any class of queries $\mathcal{C}$ and root preserving queries Q_1 and Q_2 in $\mathcal{C}$ with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$*,*

1. $Q_1 \preccurlyeq_2 Q_2 \implies Q_1 \preccurlyeq_1 Q_2$
2. $Q_1 \preccurlyeq_2 Q_2 \implies Q_1 \preccurlyeq_3 Q_2$
3. $(Q_1 \preccurlyeq_1 Q_2 \wedge Q_1 \preccurlyeq_3 Q_2) \not\Rightarrow Q_1 \preccurlyeq_2 Q_2$
4. $Q_1 \preccurlyeq_2 Q_2 \not\Rightarrow Q_1 \preccurlyeq_{2,MSO} Q_2$.

Proof.

1. This is Proposition 4.10.

2. Let $Q_1 \preccurlyeq_2 Q_2$. Let $\mathcal{Q}$ be a Boolean query and t in $\mathrm{dom}(Q_1)$ such that $\mathit{Certain}_{Q_1}(\mathcal{Q}; \mathcal{V}\mathit{iew}\,(Q_1, t))$. Let t_0 be a tree such that $\mathcal{V}\mathit{iew}\,(Q_2, t) \simeq \mathcal{V}\mathit{iew}\,(Q_2, t_0)$. There exists a tree t' with $t' \simeq t_0$ and $\mathcal{V}\mathit{iew}\,(Q_2, t') = \mathcal{V}\mathit{iew}\,(Q_2, t)$, because we considered queries closed under isomorphism. We have $Q_2(t) = Q_2(t')$ and $Q_1 \preccurlyeq_2 Q_2$, so $Q_1(t) = Q_1(t')$ by definition, which implies $\mathcal{V}\mathit{iew}\,(Q_1, t) \simeq \mathcal{V}\mathit{iew}\,(Q_1, t_0)$. We have proved $Q_1 \preccurlyeq_3 Q_2$.

3. Let D be the DTD defined by `r → a*baa*`, let $\chi_1 = \Downarrow[\mathsf{self}::a$ and $\Leftarrow::b]$ and $\chi_2 = \Downarrow::a$. Let Q_1 be the query that synthesizes validation against D and $\mathcal{X}\mathit{Reg}$ expression χ_1 and Q_2 be the query that synthesizes validation against D and $\mathcal{X}\mathit{Reg}$ expression χ_2. Those queries satisfy: $(Q_1 \preccurlyeq_1 Q_2 \wedge Q_1 \preccurlyeq_3 Q_2)$ but $Q_1 \not\preccurlyeq_2 Q_2$.

4. We show a stronger result actually; we prove that determinacy for simple annotations does not imply the existence of an *MSO* query rewriting even when $\mathcal{V}\mathit{iew}\,(Q_2, D)$ is regular.

   Let D be the DTD defined by `r → ara + ar'a` and `r' → a`. Let ann and ann_1 be the annotations defined by $\mathsf{ann}(r, r) = \mathsf{ann}(r, r') = \mathsf{false}$ and $\mathsf{ann}(r, a) = \mathsf{ann}(r', a) = \mathsf{true}$, while $\mathsf{ann}_1(r, r) = \mathsf{ann}_1(r, r') = \mathsf{ann}_1(r, a) = \mathsf{false}$ and $\mathsf{ann}_1(r', a) = \mathsf{true}$. Let $Q_1 = Q_{(D,\mathsf{ann}_1)}$ and $Q_2 = Q_{(D,\mathsf{ann})}$. Language $\mathcal{V}\mathit{iew}\,(Q_2, L(D))$ consists of all trees of depth one with nodes labeled a below root r, in odd number. Any query Q such that $\mathsf{Rewrite}(Q, Q_2) = Q_1$ would have to select the middle "a" element in a^{2n+1}, which is beyond the power of *MSO* queries. □

We define the decision problems associated to the comparisons. Those problems are parameterized by the comparison $i \in \{1, 2, 3\}$ and a class of queries $\mathcal{C}$, and prove that deciding comparisons $\preccurlyeq_2$ and $\preccurlyeq_3$ (determinacy) are at least as hard as deciding comparison $\preccurlyeq_1$ (inclusion)

Problem: Comparison $\preccurlyeq_i$ (for $\mathcal{C}$)

Input: Queries $Q_1, Q_2 \in \mathcal{C}$ with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$.
Question: $Q_1 \preccurlyeq_i Q_2$?

Proposition 4.14. *For all classes of queries $\mathcal{C} \in \{\mathcal{X}Reg, MSO\}$ there is a polynomial time reduction from comparison $\preccurlyeq_1$ (for $\mathcal{C}$) to comparison $\preccurlyeq_2$ (also for $\mathcal{C}$), and a polynomial time reduction from comparison $\preccurlyeq_1$ to comparison $\preccurlyeq_3$.*

Proof. Let $\mathcal{C}$ denote one of $\mathcal{X}Reg$ or MSO, and let Q_1 and Q_2 be two root preserving queries from $\mathcal{C}$ with identical domain D. We denote by Σ' the new alphabet: $\Sigma' = ((\Sigma\backslash\{r\}) \times \{1,2\}) \cup \{\$\} \cup \{(r,1)\}$ where r is the label of the root of trees in D. Intuitively, the \$ will be used as a tag that marks the positions selected by Q_1, while the substitution with two copies of each letter will be necessary only for the reduction to comparison $\preccurlyeq_3$.

We define a transformation τ that adds a \$ symbol as the leftmost child of every node of the trees in D: $\forall a \in \Sigma, \tau(a(t_1,t_2,\dots,t_n)) = a(\$,\tau(t_1),\dots,\tau(t_n))$. We also define morphism ϕ from Σ' to $\Sigma \cup \{\$\}$ that projects the labels on their first component. Formally, $\phi(\$) = \$$, $\phi((r,1)) = r$, and for all a in $\Sigma\backslash\{r\}, \phi((a,1)) = \phi((a,2)) = a$. Finally, D' is defined as $\phi^{-1}(\tau(D))$.

Given any tree $t \in D'$, $\tau^{-1}(t)$ returns the tree obtained from t by removing the \$ nodes (only leaves may be labeled by a \$), and $\phi(\tau^{-1}(t))$ additionally projects the labels on the first component. We define two queries Q_1' and Q_2' as follows. For every $i \in \{1,2\}$, $Q_i'(t) \cap N_{\tau^{-1}(t)} = Q_i(\phi(\tau^{-1}(t)))$, and Q_1' selects no node with label \$ whereas Q_2' selects a node with label \$ if and only if its parent node is selected by Q_1'. Queries Q_1' and Q_2' in $\mathcal{C}$ can clearly be defined in polynomial time from Q_1 and Q_2. To conclude the proof we observe that: $Q_1 \preccurlyeq_1 Q_2 \iff Q_1' \preccurlyeq_2 Q_2' \iff Q_1' \preccurlyeq_3 Q_2'$.

Here is a proof for the observation: if $Q_1 \preccurlyeq_1 Q_2$ does not hold, then there exists a tree t' and node $n \in N_t$ such that $n \in Q_1(t')\backslash Q_2(t')$. Let t_1 be a tree such that $\phi(\tau^{-1}(t_1)) = t'$ and $lab_{t_1}(n) = (a,1)$, and t_2 be obtained from t_1 by relabeling n with $(a,2)$. From $Q_2'(t_1)$ one cannot guess if the label of n is $(a,1)$ or $(a,2)$: $\mathcal{V}iew\,(Q_2',t_1) \simeq \mathcal{V}iew\,(Q_2',t_2)$, and yet $\mathcal{V}iew\,(Q_1',t_1) \not\simeq \mathcal{V}iew\,(Q_1',t_2)$. Therefore, $Q_1' \preccurlyeq_3 Q_2'$ implies $Q_1 \preccurlyeq_1 Q_2$. When $Q_1 \preccurlyeq_1 Q_2$, $Q_1' \preccurlyeq_{2,\mathcal{C}} Q_2'$ obviously holds, since in that case we only need to select in the view for Q_2' the nodes having a child labeled \$ to get the nodes selected by Q_1'. Moreover, $Q_1' \preccurlyeq_{2,\mathcal{C}} Q_2'$ implies $Q_1' \preccurlyeq_3 Q_2'$ by Proposition 4.13. □

We observe that we have used $\preccurlyeq_{2,\mathcal{C}}$ instead of $\preccurlyeq_2$ in the last paragraph of the proof, which yields the additional result that $Q_1 \preccurlyeq_1 Q_2 \iff Q_1' \preccurlyeq_{2,\mathcal{C}} Q_2'$. Consequently we also have a reduction from comparison $\preccurlyeq_1$ to the problem of deciding comparison $\preccurlyeq_{2,\mathcal{C}}$.

Example 4.5. *Figure 4.3 illustrates the reduction for two $\mathcal{X}Reg$ queries. Clearly, the queries Q_1 and Q_2 from that figure satisfy $Q_1 \preccurlyeq_1 Q_2$. Therefore, queries Q_1' and Q_2' satisfy $Q_1' \preccurlyeq_3 Q_2'$ and even $Q_1' \preccurlyeq_{2,\mathcal{X}Reg} Q_2'$. Query $\Downarrow^*$::[$\Downarrow$::\$] is a rewriting of Q_1' in terms of Q_2'.*

$$\begin{array}{ll} Q_2 = \Downarrow^*{::}a/\Downarrow^*{::}b & Q'_2 = \Downarrow^*{::}[\mathsf{self}{::}(a,1)\ \mathsf{or}\ \mathsf{self}{::}(a,2)]/\Downarrow^*{::}[\mathsf{self}{::}(b,1)\ \mathsf{or}\ \mathsf{self}{::}(b,2)] \\ & \quad \cup\, Q'_1/\Downarrow{::}\$ \\ Q_1 = \Downarrow^*{::}a/\Downarrow{::}b & Q'_1 = \Downarrow^*{::}[\mathsf{self}{::}(a,1)\ \mathsf{or}\ \mathsf{self}{::}(a,2)]/\Downarrow{::}[\mathsf{self}{::}(b,1)\ \mathsf{or}\ \mathsf{self}{::}(b,2)] \end{array}$$

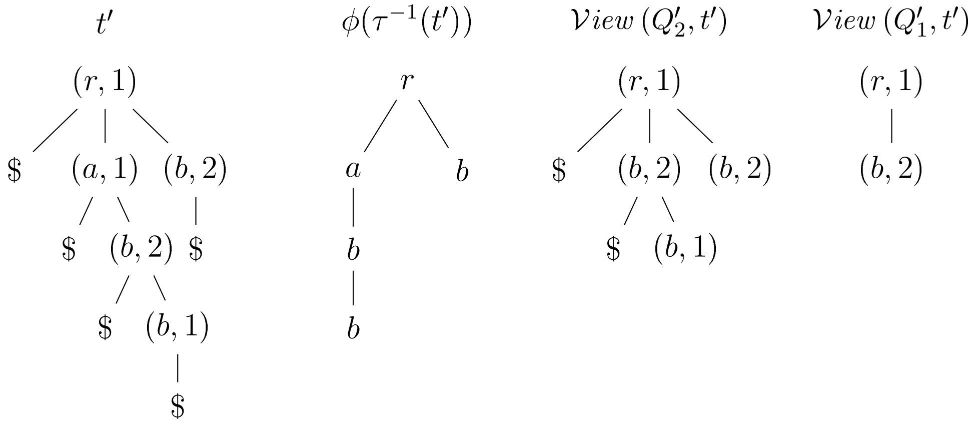


Figure 4.3.: Reduction from $\preccurlyeq_1$ to $\preccurlyeq_{2,\mathcal{X}Reg}$ and $\preccurlyeq_3$ for particular Q_1 and Q_2.

When the class of queries $\mathcal{C}$ is expressive enough, for instance when $\mathcal{C}$ is one of $\mathcal{X}Reg$ or *MSO*, we can reduce determinacy (with identifiers) to the third comparison.

Proposition 4.15. *Let $\mathcal{C}$ denote one of $\mathcal{X}Reg$ or MSO, and let Q_1, Q_2 denote two root preserving queries in $\mathcal{C}$, satisfying* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$. *We can compute in polynomial time two queries Q'_1 and Q'_2 in $\mathcal{C}$ such that $Q_1 \preccurlyeq_2 Q_2 \iff (Q_1 \preccurlyeq_1 Q_2 \wedge Q'_1 \preccurlyeq_3 Q'_2)$.*

Proof (outline). Fix $\mathcal{C} \in \{\mathcal{X}Reg, MSO\}$, and root preserving queries Q_1, Q_2 such that $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$. We first test the inclusion, and then we must check not only isomorphism constraints, but also that "the same nodes appear at the same position". For this purpose we modify the domain D before we test comparison $\preccurlyeq_3$, inserting dummy nodes into the first view so as to indicate the positions.

Formally, the proof works as follows: let \$ represent a new symbol outside Σ. The alphabet of D' is $\Sigma \cup \{\$\}$, and in a tree of D' every node of odd depth is labeled with a \$ and has a unique child, and every node of even depth has label in Σ. The new domain D' contains for each tree t in D the tree t' obtained from t by the transformation ϕ replacing every subtree $a(t_1, \ldots, t_n)$ by $a(\$(t_1, \ldots, t_n))$. Transformation ϕ is clearly bijective, and we also observe that if D is expressible in $\mathcal{C}$, D' is also expressible in $\mathcal{C}$. Next, we define two queries Q'_1 and Q'_2 of domain D' such that for all $t' \in D'$, $Q'_2(t') = Q_2(\phi^{-1}(t'))$, i.e., Q'_2 hides all \$-labeled nodes and the nodes hidden by Q_2 in t, and $Q'_1(t')$ contains exactly $Q_1(\phi^{-1}(t'))$ plus every node n with label \$ whose child belongs to $Q'_2(t')$. It is easy to build such queries in polynomial time from Q_1 and Q_2, for $\mathcal{X}Reg$ as well as for query automata. Thus, we have constructed two queries Q'_1 and Q'_2 such that $Q_1 \preccurlyeq_2 Q_2 \iff (Q_1 \preccurlyeq_1 Q_2 \wedge Q'_1 \preccurlyeq_3 Q'_2)$. At first glance, this looks like a Turing reduction, because we use two instances of $\preccurlyeq_3$: one for $Q'_1 \preccurlyeq_3 Q'_2$ and one for $Q_1 \preccurlyeq_1 Q_2$ (we recall from Proposition 4.14 that comparison $\preccurlyeq_1$ reduces into $\preccurlyeq_3$). However, it is easy to build a single instance from these two: we can use disjoint alphabets for the two instances by copying the alphabet, and then use as domain the set of trees whose root has two children; each child being devoted to one instance. □

Example 4.6. *Figure 4.4 illustrates the reduction. Clearly, the queries Q_1 and Q_2 from that figure satisfy $Q_1 \preccurlyeq_1 Q_2$, and also $Q_1 \preccurlyeq_2 Q_2$: one can even rewrite Q_1 in terms of Q_2 using query $\Downarrow^*::a/\Downarrow::b$. This is witnessed by $Q'_1 \preccurlyeq_3 Q'_2$.*

$Q_2 = \Downarrow^*::a/[\Downarrow::b \text{ and } \Uparrow::c] \cup \Downarrow^*::b$ $\qquad$ $Q'_2 = \Downarrow^*::a/[\Downarrow::b \text{ and } \Uparrow::c] \cup \Downarrow^*::b$

$Q_1 = \Downarrow^*::c/\Downarrow::a/\Downarrow::b$ $\qquad$ $Q'_1 = \Downarrow^*::c/\Downarrow::\$/\Downarrow::a/\Downarrow::\$/\Downarrow::b$

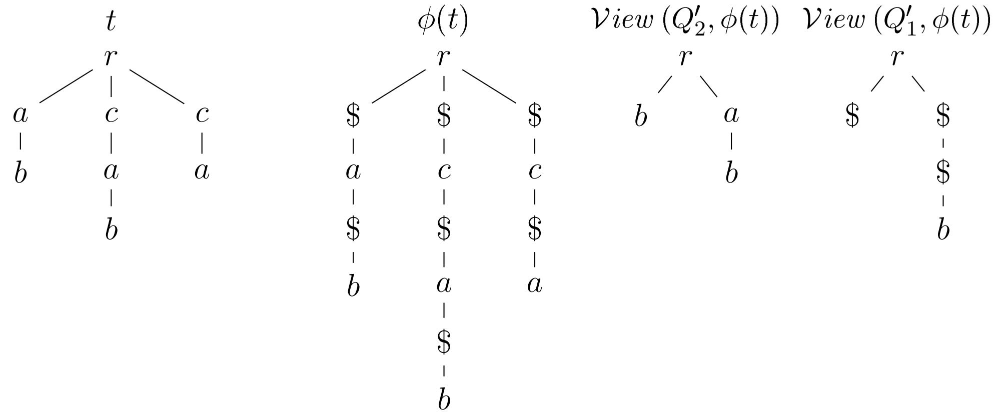

t
$\phi(t)$
$\mathcal{V}iew\,(Q'_2, \phi(t))$
$\mathcal{V}iew\,(Q'_1, \phi(t))$
r
a
c
b
\$

Figure 4.4.: Reduction from $\preccurlyeq_2$ to $\preccurlyeq_3$ for particular Q_1 and Q_2.

Complexity of Inclusion Deciding if $Q_1 \preccurlyeq_1 Q_2$ is EXPTIME-complete for Regular XPath queries: we already mentioned this complexity for the containment of Regular XPath queries in Section 3.3, and since this holds for Boolean queries, the interval-boundedness restriction does not help. We will show that this complexity can be lowered to PSPACE over non-recursive DTDs.

Deciding if $Q_1 \preccurlyeq_1 Q_2$ is in PTIME for query automata. We observe that this polynomial complexity stems from our representation via maximal languages and from the hypothesis that both automata have domain D. These conventions make the Exptime-hardness of tree automata inclusion irrelevant for the verification of $\preccurlyeq_1$.

4.2.3. Undecidability Results for Comparisons $\preccurlyeq_2$ and $\preccurlyeq_3$.

Theorem 4.16. *Given $\mathcal{C} \in \{\mathcal{X}Reg, MSO\}$, and two root preserving queries Q_1 and Q_2 in $\mathcal{C}$, testing $Q_1 \preccurlyeq_{2,\mathcal{C}} Q_2$ is undecidable.*

Proof. We use a reduction from regular separability of context-free grammars. Recall that two context-free grammars G_1 and G_2 over the alphabet Γ are *regularly separable* if there exists a regular language R (over Γ) such that $L(G_1) \subseteq R$ and $L(G_2) \subseteq R^{\complement}$, where $R^{\complement}$ is the complement of R. Checking regular separability of two context-free languages is known to be undecidable [SW73].

We give the proof for $\mathcal{C} = \mathcal{X}Reg$; the result for MSO follows the same lines. The reduction constructs a DTD D defining the set of all derivation trees of G_1 and G_2. The query Q_2 hides all nonterminals from the derivation tree except the root. The nodes selected by Q_2 are the yield of the trees in D, and they form a word of $L(G_1) \cup L(G_2)$. The query Q_1 works similarly except that it also hides terminals derived from nonterminals of G_2; essentially, it returns only words of $L(G_1)$.

If G_1 and G_2 are separable by a regular set R, then the regular expression describing R can be easily rewritten into a $\mathcal{X}Reg$ query Q such that for all t in D, $Q(\mathcal{V}iew\,(Q_2, t)) = Q_1(t)$, that is $Q_{(D,Q_1)} \preccurlyeq_{2,\mathcal{C}} Q_{(D,Q_2)}$. Conversely, suppose there is a $\mathcal{X}Reg$ query Q such that for all t in D, $Q(\mathcal{V}iew\,(Q_2, t)) = Q_1(t)$. Essentially, Q selects words from $L(G_1)$ and hides words from $L(G_2)$, hence it separates G_1 and G_2. Then Q is equivalent to a tree MSO formula φ [B6̈0], and we remark that φ is interpreted on trees of height one only. Therefore, there exists a word MSO formula ψ that captures exactly the words consisting of labels of the consecutive children of the root node. This formula ψ can be converted into a regular expression [TW68] which defines a set separating G_1 and G_2. □

We prove similarly that determinacy is undecidable:

Theorem 4.17. *Given two root preserving $\mathcal{X}Reg$ queries Q_1 and Q_2, testing $Q_1 \preccurlyeq_2 Q_2$ is undecidable.*

Proof. The proof is similar to the one for Theorem 4.16, hiding derivations of context-free grammars, except that the reduction is toward emptiness of intersection: we recall that the problem of deciding whether $L(G_1) \cap L(G_2) = \varnothing$ given two context-free grammars G_1 and G_2, is undecidable. □

Proposition 4.18. *We denote by* $\equiv_3$ *the equivalence relation* $Q_1 \equiv_3 Q_2 \iff Q_1 \preccurlyeq_3 Q_2 \wedge Q_2 \preccurlyeq_3 Q_1$. *In general (and even if the visibility of a node depends only on its label) testing whether* $Q_1 \equiv_3 Q_2$ *is undecidable, therefore testing whether* $Q_1 \preccurlyeq_3 Q_2$ *is undecidable.*

Proof. Given an instance of PCP $\mathcal{P} : u_1, \dots u_n, v_1, \dots v_n$ with $u_i, v_i \in \Sigma^*$ for all $i \leqslant n$, we define as follows a DTD D over alphabet $\Sigma \cup \{\mathtt{u}, \mathtt{v}, \#, \mathtt{1}, \dots \mathtt{n}\}$, together with access functions X_1, X_2. The DTD production rules are: $\mathtt{r} \to \mathtt{u} \mid \mathtt{v}$, $\mathtt{u} \to (\mathtt{u_1}, \mathtt{u}, \mathtt{1}) \mid \dots \mid (\mathtt{u_n}, \mathtt{u}, \mathtt{n}) \mid \#$, and $\mathtt{v} \to (\mathtt{v_1}, \mathtt{v}, \mathtt{1}) \mid \dots \mid (\mathtt{v_n}, \mathtt{v}, \mathtt{n}) \mid \#$, and the access functions are, for all j in $\{1, 2\}$, i in $\{1, \dots n\}$, and $\alpha \in \Sigma \cup \{\#\}$:

$\mathsf{ann}_1(r, u) = \mathsf{ann}_1(r, v) = \mathsf{false}$,
$\mathsf{ann}_2(r, u) = \mathsf{ann}_2(r, v) = \mathsf{true}$,
$\mathsf{ann}_j(u, u) = \mathsf{ann}_j(v, v) = \mathsf{false}$, and
$\mathsf{ann}_j(u, \alpha) = \mathsf{ann}_j(v, \alpha) = \mathsf{ann}_j(u, i) = \mathsf{ann}_j(v, i) = \mathsf{true}$

The view for annotation ann_1 consists of some tree of depth one, and therefore can be identified with words. See Figure 4.5 for an illustration of the PCP instance ($u_1 = aab, u_2 = ba, u_3 = b$, $v_1 = aa, v_2 = bb, v_3 = abb$) over alphabet $\Sigma = \{a, b\}$: the two annotations derived from this instance do not satisfy $Q_{(D,\mathsf{ann}_1)} \equiv_3 Q_{(D,\mathsf{ann}_2)}$. $Q_{(D,\mathsf{ann}_1)}(t)$ can easily be obtained from $Q_{(D,\mathsf{ann}_2)}$ by erasing u or v, so $Q_{(D,\mathsf{ann}_1)} \preccurlyeq_3 Q_{(D,\mathsf{ann}_2)}$ trivially holds. Clearly, $Q_{(D,\mathsf{ann}_2)}(t) \preccurlyeq_3 Q_{(D,\mathsf{ann}_1)}(t)$ if and only if there is no solution to the PCP problem. Hence, $Q_{(D,\mathsf{ann}_1)} \equiv_3 Q_{(D,\mathsf{ann}_2)}$ if and only if the answer of $\mathcal{P}$ is negative. Thus testing $Q_{(D,\mathsf{ann}_1)} \equiv_3 Q_{(D,\mathsf{ann}_2)}$ is undecidable. □

4.2.4. Determinacy for *MSO*

For interval-bounded views, comparisons $\preccurlyeq_{2,MSO}$ and $\preccurlyeq_2$ are equivalent:

Proposition 4.19. *Let* Q_1 *and* Q_2 *denote two query automata, with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$ *and* Q_2 *interval-bounded. Then* $Q_1 \preccurlyeq_{2,MSO} Q_2$ *if and only if* $Q_1 \preccurlyeq_2 Q_2$. *Furthermore, if* Q_2 *is* k*-interval-bounded and* $Q_1 \preccurlyeq_{2,MSO} Q_2$, *one can compute a query automaton* Q *such that* $\mathsf{Rewrite}(Q, Q_2) = Q_1$ *in time exponential in* k.

Proof. Since $Q_1 \preccurlyeq_{2,MSO} Q_2 \implies Q_1 \preccurlyeq_2 Q_2$, we only need to prove that one can compute a query automaton Q such that $\mathsf{Rewrite}(Q, Q_2) = Q_1$ whenever $Q_1 \preccurlyeq_2 Q_2$. Let $k \in \mathbb{N}$ a natural number such that Q_2 is k-interval-bounded. We suppose $Q_1 \preccurlyeq_2 Q_2$. Then, by Proposition 4.13, $Q_1 \preccurlyeq_1 Q_2$. We define an automaton $\mathcal{A} = (\Sigma \times \Sigma_\varepsilon^2, S, \Gamma, I, F, R)$ with language $L(\mathcal{A}) = L_{Q_1 \otimes Q_2}$. Note that since we suppose $Q_1 \preccurlyeq_1 Q_2$, no label (a, a, ε) can occur in any tree accepted by $\mathcal{A}$. Next, we abstract from elements in t that are not selected by Q_2 in order to rewrite Q_1 in terms of Q_2. For this, we use the same construction as in Proposition 4.2 which computes an automaton for the

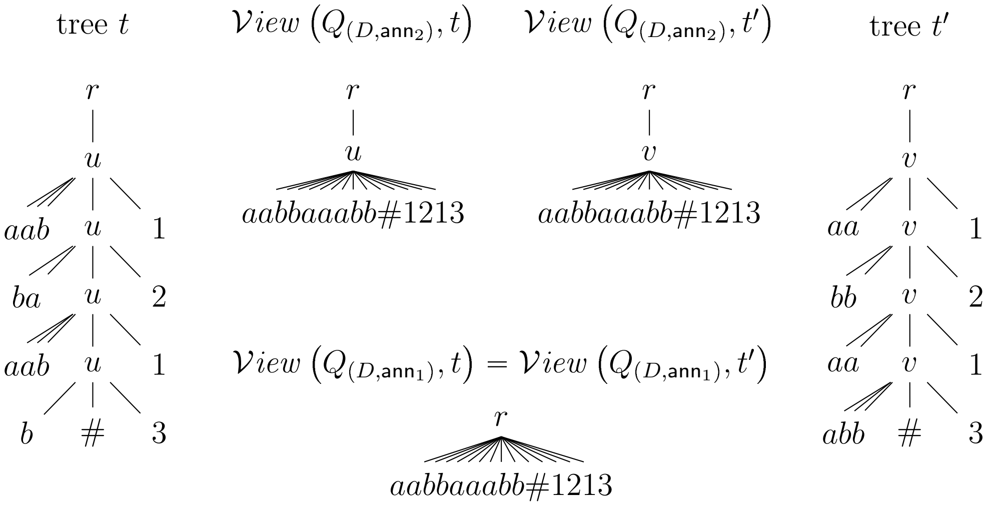

tree t
$\mathcal{V}iew\left(Q_{(D,\mathsf{ann}_2)}, t\right)$
$\mathcal{V}iew\left(Q_{(D,\mathsf{ann}_2)}, t'\right)$
tree t'
r
u
aab
u
1
ba
u
2
aab
u
1
b
#
3
r
u
$aabbaaabb\#1213$
r
v
$aabbaaabb\#1213$
$\mathcal{V}iew\left(Q_{(D,\mathsf{ann}_1)}, t\right) = \mathcal{V}iew\left(Q_{(D,\mathsf{ann}_1)}, t'\right)$
r
$aabbaaabb\#1213$
r
v
aa
v
1
bb
v
2
aa
v
1
abb
#
3

Figure 4.5.: PCP encoding for comparison $\preccurlyeq_3$.

view. Indeed, $\mathcal{A}$ can be considered as defining an interval bounded query on trees labeled by $\Sigma \times \Sigma_\varepsilon$ which will select all the nodes labeled by $\Sigma \times \Sigma$ as $Q_1 \preccurlyeq_1 Q_2$.

Construction of an automaton rewriting Q_1 in terms of Q_2: the idea is to eliminate transitions $q \xrightarrow{(\eta,(a,\varepsilon,\varepsilon)):\gamma} q'$ for every $q, q' \in S, \eta \in \{op, cl\}, q \in \Sigma, \gamma \in \Gamma$, replacing them with ϵ-transitions. The interval-boundedness restriction allows us to eliminate those transitions. First, let $\mathcal{E} \subseteq S \times S$ be the set of all pairs (q, q') such that $\mathcal{A}$ accepts some tree with labels in $\Sigma \times \{\varepsilon\} \times \{\varepsilon\}$ from initial state q to final state q'. More formally, $(q, q') \in \mathcal{E}$ if and only if there is some tree t in $L(\mathcal{A}_{q,q'})$ satisfying $lab_t(n) \in \Sigma \times \{\varepsilon\} \times \{\varepsilon\}$ for all $n \in N_t$.

We define a VPA $\mathcal{B} = (\Sigma \times \Sigma_\varepsilon, S \times \Gamma^{\leqslant k}, \Gamma^{\leqslant k} \times \Gamma, I \times \{\epsilon\}, F \times \{\epsilon\}, R')$ from $\mathcal{A}$ with the following rules. Basically, the VPA $\mathcal{B}$ simulates within its states a stack of depth at most k.

- $\mathcal{B}$ has transition $(q, u) \xrightarrow{\epsilon} (p, u\gamma)$ for every transition $q \xrightarrow{(op,(a,\varepsilon,\varepsilon)):\gamma} p$ of $\mathcal{A}$ and $u \in \Gamma^{\leqslant (k-1)}$.
- $\mathcal{B}$ has transition $(q, u\gamma) \xrightarrow{\epsilon} (p, u)$ for every transition $q \xrightarrow{(cl,(a,\varepsilon,\varepsilon)):\gamma} p$ of $\mathcal{A}$ and $u \in \Gamma^{\leqslant (k-1)}$.
- $\mathcal{B}$ has transition $(q, u) \xrightarrow{(op,(a,x_1)):\langle u,\gamma\rangle} (p, \epsilon)$ for every transition $q \xrightarrow{(op,(a,x_1,a)):\gamma} p$ of $\mathcal{A}$ and $u \in \Gamma^{\leqslant k}$.

- $\mathcal{B}$ has transition $(q,\epsilon) \xrightarrow{(cl,(a,x_1)):\langle u,\gamma\rangle} (p,u)$ for every transition $q \xrightarrow{(cl,(a,x_1,a)):\gamma} p$ of $\mathcal{A}$ and $u \in \Gamma^{\leqslant k}$.
- $\mathcal{B}$ has transition $(q,u) \xrightarrow{\epsilon} (p,u)$ for every $u \in \Gamma^{\leqslant k}$ and $(q,p) \in \mathcal{E}$.

One can compute $\mathcal{B}$ from $\mathcal{A}$ in time exponential in k. There is a polynomial p_1 such that $|\mathcal{B}| \leqslant (p_1(|Q_1| \times |Q_2|))^k$. To conclude the proof, we observe that due to our determinacy hypothesis, the language accepted by $\mathcal{B}$ is maximal, and by construction, it defines a query Q such that $\mathsf{Rewrite}(Q,Q_2) = Q_1$, as evidenced by the following invariant.

Let w a word over $\{op,cl\} \times \Sigma$, $\langle q,u\rangle$ a state in $S \times \Gamma^{\leqslant k}$, and σ a word over $\Gamma \times \Gamma^{\leqslant k}$. For the sake of clarity, we denote by σ' the same σ considered as a word over Γ. We claim that for all such w, q, and σ, $\mathcal{B}$ preserves the following invariant.

> **Invariant:** *$\mathcal{B}$ can reach configuration $(\langle q,u\rangle,\sigma)$ after reading w if and only if there exists a word w' over $\{op,cl\} \times \Sigma \times \Sigma_\varepsilon \times \Sigma_\varepsilon$ such that the following two conditions are satisfied: (1) $\pi_{2,3}(w') = w$, and (2) $\mathcal{A}$ can reach configuration $(q,\sigma' u)$ after reading w'.*

□

Remark 4.6. *There is no way round the exponential blowup: for every $n \geqslant 0$ there exist n-interval-bounded query automata Q_1 and Q_2 of size $O(n)$, such that no automaton Q satisfying* $\mathsf{Rewrite}(Q,Q_2) = Q_1$ *has size less than 2^n.*

Proof. We prove this remark with a simple counting argument. Consider the DTD $D_n : r \to a_0\#a_0$ and, for every $i < n$, $a_i \to a_{i+1}a_{i+1}$. This DTD D_n describes a single tree t_n with yield $(a_n)^{2^n}\#(a_n)^{2^n}$. Let Q_2 be the query with domain $\{t_n\}$ that selects the leaves a_n, and Q_1 the query with domain $\{t_n\}$ that selects the $(2^n)^{\text{th}}$ leaf a_n in document order. Q_1 and Q_2 can clearly be represented by VPAs (query automata) of linear size $O(n)$. However, $\mathcal{V}iew\,(Q_2,t_n)$ is a tree of depth one with yield $(a_n)^{2^{n+1}}$. Therefore, a query automaton Q satisfying $\mathsf{Rewrite}(Q,Q_2) = Q_1$ must select the letter at position 2^n in the word $(a_n)^{2^{n+1}}$. Obviously, this cannot be achieved by query automata having fewer than 2^n states. □

From this proposition, since determinacy is co-recursively enumerable, and $\leqslant_{2,MSO}$ is recursively enumerable, we can deduce immediately the decidability of $\leqslant_2$ for interval-bounded annotations, but we can do much better. A first approach for testing $\leqslant_2$ could be to build the "square" of $\mathcal{B}$ and test whether there are two trees $t \neq t'$ accepted by $\mathcal{B}$, with the same projection π_1; $\pi_1(t) = \pi_1(t')$. We can test this property on $\mathcal{B}$, in terms of accessibility of states.

Corollary 4.20. *Let Q_1 and Q_2 denote two query automata, with* $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2)$. *Given a fixed constant k, we can test in polynomial time whether*

$Q_1 \preccurlyeq_2 Q_2$ for Q_2 k-interval bounded. This holds in particular for upward closed views.

Similarly, when the depth of the domain is bounded by a fixed constant, the complexity for testing $Q_1 \preccurlyeq_2 Q_2$ becomes NLogspace.

Proof. We first check in polynomial time that $Q_1 \preccurlyeq_1 Q_2$; otherwise, $Q_1 \not\preccurlyeq_2 Q_2$. We then build the automaton $\mathcal{B}$ above, and eliminate its epsilon transitions, resulting in a VPA $(\Sigma \times \Sigma_\varepsilon, S_B, \Gamma_B, I_B, F_B, R_B)$. Let $\mathcal{B}_{\text{square}}$ denote the square of this automaton $\mathcal{B}$, namely $(\Sigma \times \Sigma_\varepsilon \times \Sigma_\varepsilon, S_B^2, \Gamma_B^2, I_B^2, F_B^2, R_{\text{square}})$ such that $\mathcal{B}_{\text{square}}$ has rule $(q_1, q_2) \xrightarrow{(\eta,(b,\alpha_1,\alpha_2)):(\gamma_1,\gamma_2)} (q_1', q_2') \in R_{\text{square}}$ iff $\mathcal{B}$ has rules $q_1 \xrightarrow{(\eta,(b,\alpha_1)):\gamma_1} q_1' \in R$ and $q_2 \xrightarrow{(\eta,(b,\alpha_2)):\gamma_2} q_2' \in R$. We could alternatively drop the first component of the letters (b) without any consequences for the remaining of the proof. By construction, and as we supposed $Q_1 \preccurlyeq_1 Q_2$, it holds that $Q_1 \preccurlyeq_2 Q_2$ if and only if for all b, α_1, α_2 with $\alpha_1 \neq \alpha_2$, the language of $\mathcal{B}_{\text{square}}$ contains no tree with a node labeled (b, α_1, α_2). This is a problem of reachability, which can be solved in polynomial time for VPAs. For instance, we modify $\mathcal{B}_{\text{square}}$ so that its state remembers if a letter of the form (b, α_1, α_2) with $\alpha_1 \neq \alpha_2$ has already been read: the resulting automaton $\mathcal{B}_1$ has states, initial and final states $S_B^2 \times \{0, 1\}$, $I_B^2 \times \{0\}$ and $F_B^2 \times \{1\}$ respectively. Moreover, $\mathcal{B}_1$ has transition $(q_1, q_2, x) \xrightarrow{(\eta,(b,\alpha_1,\alpha_2)):\gamma_2} q_1', q_2', y$ if and only if the following two conditions are satisfied: (1) $(q_1, q_2) \xrightarrow{(\eta,(b,\alpha_1,\alpha_2)):(\gamma_1,\gamma_2)} (q_1', q_2') \in R_{\text{square}}$ and (2) $y = 1$ if and only if $x = 1$ or $\alpha_1 \neq \alpha_2$, otherwise $y = 0$. Since there is a polynomial p_2 such that $\mathcal{B}_1$ is built in time at most $(p_2(|Q_1| \times |Q_2|))^k$, we get the polynomial time complexity when k is a fixed constant.

When the depth of the domain is bounded by a fixed constant k, Proposition 3.13 proves that $\mathcal{B}_1$ accepts a tree of size polynomial or accepts no tree at all. Consequently, we can guess non-deterministically a tree of polynomial size and guess a run of the VPA $\mathcal{B}_1$ over this word. We cannot afford to build the full $\mathcal{B}_1$, but it can be evaluated on-the-fly: only a counter and the current stack and state of the VPA need to be stored, which requires only logarithmic space (the stack has constant depth). This way we can test emptiness of the VPA in NLogspace. One could alternatively prove that emptiness of $\mathcal{B}_1$ can be evaluated in NLogspace by reduction to emptiness for word automata: when the depth of the domain is bounded by a constant, $\mathcal{B}_1$ can be seen as an automaton of polynomial size, of which the transitions can still be evaluated on the fly. This guarantees we can test its emptiness in NLogspace. □

However, the full construction of $\mathcal{B}$ induces an exponential cost in terms of time and space, so that for general interval-bounded queries, the above approach uses exponential space. We provide a polynomial space algorithm instead for interval-bounded queries.

Lemma 4.21. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two query automata, expressing queries Q_1 and Q_2, such that Q_2 is interval bounded. If there are two trees t, t' such that $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ but $Q_1(t) \neq Q_1(t')$, then there are two such trees of size exponential and depth polynomial in the size of the automata $\mathcal{A}_1, \mathcal{A}_2$.*

Proof. We prove this with a rough pumping argument: the purpose of this lemma being to obtain a polynomial space algorithm in Proposition 4.22, little care has been given to lower the degree of the polynomial. We need to adapt the standard pumping argument due to the necessity to synchronize two trees and two automata instead of one tree and one automaton: we need to consider three nodes instead of two in order to preserve the difference between the views for Q_1. Let $\mathcal{A}_1 = (\Sigma, S_1, \Gamma_1, I_1, F_1, \Delta_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, \Gamma_2, I_2, F_2, \Delta_2)$ two query automata, with corresponding queries Q_1 and Q_2 such that Q_2 is an interval bounded queries and $Q_1 \preccurlyeq_1 Q_2$. Let (t, t') be a pair of trees of minimal size such that $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ but $Q_1(t) \neq Q_1(t')$. Let ρ_2^t (resp. $\rho_2^{t'}$) denote accepting runs of the automaton $\mathcal{A}_2$ on $t \otimes Q_2$ (resp. $t' \otimes Q_2$), and ρ_1^t (resp. $\rho_1^{t'}$) denote accepting runs of the automaton $\mathcal{A}_1$ on $t \otimes Q_1$ (resp. $t' \otimes Q_1$). We also denote by $(\rho_2^t)^\uparrow$, $(\rho_1^{t'})^\uparrow$, etc. the corresponding functions that map a node n to the pair of states assigned by the run to the automaton before reading the opening tag and after processing the closing tag of n, as detailed on page 62.

Vertical pumping: We decorate every node n in $Q_2(t)$ (therefore also in $Q_2(t')$) with the tuple $\rho(n) = (\rho_2^t(n), \rho_2^{t'}(n), \rho_1^t(n), \rho_1^{t'}(n))$. Suppose there is some node in $Q_2(t)$ at depth strictly greater than $(k+1) \times 2 \times |S_2|^2 \times |S_1|^2$ in t or t', then there are three distinct nodes $n^\uparrow, n^\circ, n^\downarrow$ in $Q_2(t)$ such that $n^\uparrow$ is an ancestor of n°, n° an ancestor of $n^\downarrow$, and $\rho(n^\uparrow) = \rho(n^\circ) = \rho(n^\downarrow)$ as depicted in Figure 4.6.

We consider two cases depending on whether there exists below n° a node n that belongs to $Q_1(t) \,\Delta\, Q_1(t')$. In the first case we assume there is some node n below n° that belongs to $Q_1(t) \,\Delta\, Q_1(t')$. Then we could replace the subtree below $n^\uparrow$ with the subtree below n° in t and t': the two trees thus obtained would have same view for Q_2 and different views for Q_1, which contradicts minimality of the pair (t, t'). In the second case there is no node $n \in Q_1(t) \,\Delta\, Q_1(t')$ below n°, but then we could replace the subtree below n° with the subtree below $n^\downarrow$ in t and t': the two trees thus obtained would have same view for Q_2 and different views for Q_1, which contradicts minimality of the pair (t, t'). So either way, our minimality hypothesis enters in contradiction with the existence of a node of depth greater than $(k+1) \times 2 \times |S_2|^2 \times |S_1|^2$ in $Q_2(t)$ or in $Q_2(t')$. Hence no node in $Q_2(t)$ or $Q_2(t')$ has depth greater than $(k+1) \times 2 \times |S_2|^2 \times |S_1|^2$.

Thus, t and t' have polynomial depth. Notice that the pumping argument used to bound the depth of the trees does not increase the size of the trees. We can use another pumping argument, pumping "horizontally" this time, and bound the number of children of every node in t or t' by an exponential.

Horizontal pumping: As before we use a pumping argument over nodes in $Q_2(t)$, because this makes it easier to preserve equality of the views for Q_2. Let $n \in Q_2(t)$. Then n also belongs to $Q_2(t')$. However, it could very well be that no child of n in t or t' belongs to $Q_2(t)$, while some descendant of n would still belong to $Q_2(t)$. To avoid those difficulties, we consider the children $n_1, n_2, \ldots, n_M$ of n in $\mathcal{V}iew\,(Q_2, t)$, in document order. We decorate each node n_i with two tuples $\vec{\rho}(n_i, op)$ and $\vec{\rho}(n_i, cl)$ in $(S_1 \times \Gamma_1^{\leqslant k})^2 \times (S_2 \times \Gamma_2^{\leqslant k})^2$. Tuple $\vec{\rho}(n_i, op)$ is associated to the opening tag of n_i and $\vec{\rho}(n_i, cl)$ to its closing tag. The tuples are defined as follows. Let $d_t \leqslant k$ denote the number of stack symbols that have been added (and not yet removed) after reading the opening tag of n and before reading the opening tag of n_i in t: $d_t(n_i) = depth_t(n_i) - depth_t(n) - 1$, and similarly $d_{t'}(n_i) = depth_{t'}(n_i) - depth_{t'}(n) - 1$. The tuples $\vec{\rho}(n_i, op)$ and $\vec{\rho}(n_i, cl)$ are respectively defined as $((q_2, u_2), (q'_2, u'_2), (q_1, u_2), (q'_1, u'_1))$ and $((s_2, u_2), (s'_2, u'_2), (s_1, u_2), (s'_1, u'_1))$ where $(\rho_2^t)^{\uparrow}(n_i) = (q_2, s_2)$, $(\rho_1^{t'})^{\uparrow}(n_i) = (q'_1, s'_1)$, etc. and $u_2 \in (\Gamma_2)^{d_t(n_i)}$ contains the $d_t(n_i)$ topmost symbols of the stack for run ρ_2^t before processing the opening tag of node n_i, $u'_1 \in (\Gamma_1)^{d_{t'}}$ contains the $d_{t'}(n_i)$ topmost symbols of the stack for run $\rho_1^{t'}$ before processing the opening tag of node n_i etc.

We assume that Γ_1, Γ_2 both contain at least two elements. The other cases can be treated similarly. The number of different tuples $\vec{\rho}$ that can be constructed is strictly smaller than $|S_1|^2 \times |\Gamma_1|^{2k+2} \times |S_2|^2 \times |\Gamma_2|^{2k+2}$. Hence if $M \geqslant 2|S_1|^2 \times |\Gamma_1|^{2k+2} \times |S_2|^2 \times |\Gamma_2|^{2k+2}$, there exist $1 \leqslant i < j < l \leqslant M$ such that $\vec{\rho}(n_i, op) = \vec{\rho}(n_j, op) = \vec{\rho}(n_l, op)$. This however contradicts the minimality of t and t': the trees $t_{i,j}$ and $t'_{i,j}$ obtained from t and t' by removing all tags between the opening of n_i (included) and the opening of n_j (excluded) satisfy $\mathcal{V}iew\,(Q_2, t_{i,j}) = \mathcal{V}iew\left(Q_2, t'_{i,j}\right)$, and likewise the trees $t_{j,l}$ and $t'_{j,l}$ obtained by removing all tags between n_j and n_l. The contradiction stems from the observation that $Q_1(t_{i,j}) \neq Q_1(t'_{i,j})$ or $Q_1(t_{j,l}) \neq Q_1(t'_{j,l})$. This concludes the proof that every node from $Q_2(t)$ has at most $2 \times |S_1|^2 \times |\Gamma_1|^{2k+1} \times |S_2|^2 \times |\Gamma_2|^{2k+1}$ children in $\mathcal{V}iew\,(Q_2, t)$.

We still have to bound the number of nodes in $N_t \backslash Q_2(t)$ and likewise in t', but here the pumping argument is the usual one, as we can apply the pumping argument from Proposition 3.13 independently in t and t' on the "hidden" parts, provided nodes selected by Q_2 are not affected. For each node $n \in N_t$ and every sequence $n_1, n_2, \ldots, n_L$ of consecutive children of n, if $L \geqslant |S_1| \times |S_2|$ then one of these children has necessarily a descendant in $Q_2(t)$ otherwise the pumping argument from Proposition 3.13 would contradict the minimality of t and t'. Consequently, the number of children of a node in t or t' can be roughly bounded[3] by $O(|S_1|^3 \times |\Gamma_1|^{2k+2} \times |S_2|^3 \times |\Gamma_2|^{2k+2})$. More-

[3] We chose to simplify the presentation, but one could obtain much better bounds. For instance, we use pairs of states $\rho_2^t(n_i)$ in the definition of $\vec{\rho}$, in order to apply the

over, each node $n \in N_t$ without descendant in $Q_2(t)$ has no descendants of depth greater than its own depth plus $|S_1|^2 \times |S_2|^2$, according to the pumping argument of Proposition 3.12. Therefore, no node in t or t' has depth greater than $k \times 3 \times |S_2|^2 \times |S_1|^2$. The combination of those horizontal and vertical pumping arguments allows to conclude the proof for Lemma 4.21: t and t' have size at most exponential. □

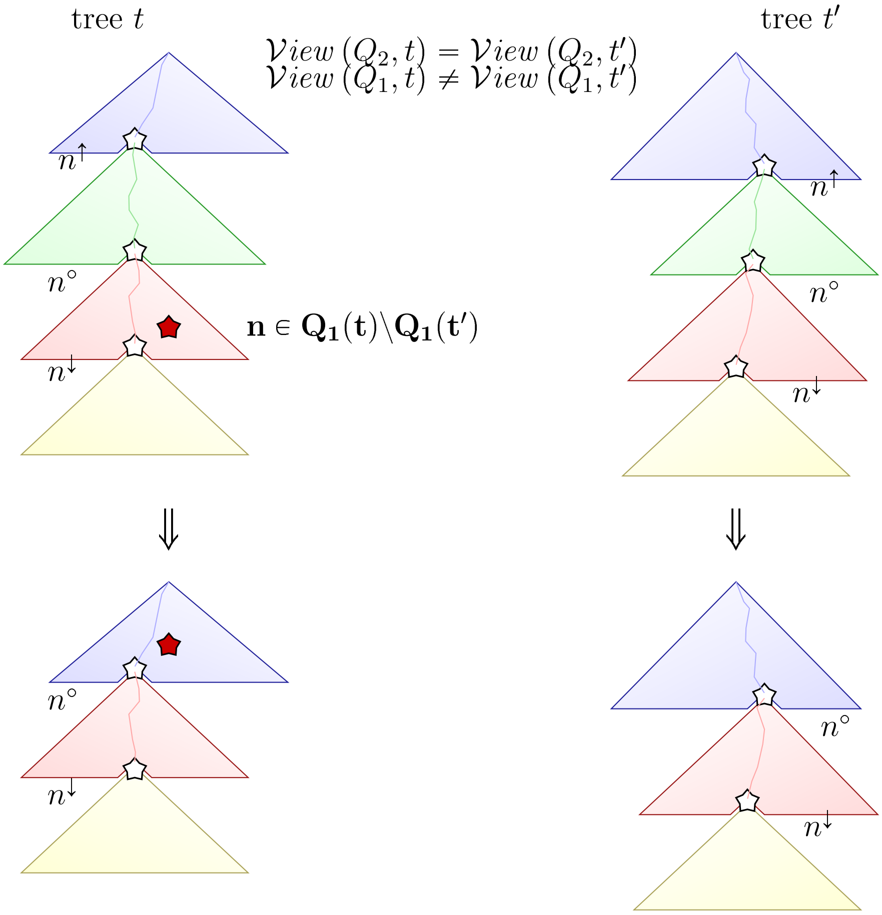


Figure 4.6.: Pumping argument for comparison $\preccurlyeq_2$.

Proposition 4.22. *Given query automata $\mathcal{A}_1$ and $\mathcal{A}_2$ expressing queries Q_1 and Q_2 with Q_2 interval bounded, we can test $Q_1 \preccurlyeq_{2,MSO} Q_2$ in polynomial space.*

pumping argument on nodes, but we could "dissociate" the opening and closing tags as in the proof of Proposition 3.13, which would lower the degree of S_1 and S_2 in our polynomials.

Proof. Let Q_1 be an *MSO* query and Q_2 an *MSO* k-interval-bounded query. Then, by Lemma 4.19 it is enough to test whether there are trees t, t' such that $Q_2(t) = Q_2(t')$ but $Q_1(t) \neq Q_1(t')$. Moreover, Lemma 4.21 gives a bound on the size and depth of t and t'. This suggests the following algorithm: we guess the size of t, t'. Those trees have exponential size, so their size can be represented using polynomial space only. Then we guess step by step the run of both view automata over t and t'. We only need to store the stack and the current state, which provides a non-deterministic algorithm in polynomial space. The result then follows from Savitch's theorem. □

In the following, we are interested in query automata with a domain equivalent to a non recursive DTD. We write that *the domain is equivalent to a non-recursive DTD* even if no DTD is manipulated here: the only property that is required is actually that every tree of the domain has depth bounded by a polynomial in the queries. As queries whose domain is equivalent to a non-recursive DTD are a special case of interval-bounded queries, we get immediately from Proposition 4.22:

Corollary 4.23. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ denote two query automata expressing queries Q_1 and Q_2. When the domain is equivalent to a non-recursive DTD, one can test $Q_1 \preccurlyeq_{2,MSO} Q_2$ in polynomial space.*

We recall that a *straight line program* is a context free grammar such that there is a single production from each non-terminal, and the production relation is acyclic. Thus, each straight line program G represents a single word w_G. In that setting, the *Compressed Membership Problem* consists in deciding given a regular expression with squares E (over alphabet T), and a word w over T given by a straight line program, whether w belongs to the language of E.

Theorem 4.24 (Theorem 6 in [Loh10]). *The Compressed Membership Problem is* PSPACE*-complete for regular expressions with squares.*

Lemma 4.25. *For query automata, comparison $\preccurlyeq_2$ is* PSPACE*-hard even when the domain is equivalent to a non-recursive DTD.*

Proof. The proof works by reduction from the compressed membership problem for regular expressions with squares. Fix a straight line program $G = (V, T, S, P)$ and a regular expression with squares E over T. We can compute in polynomial time a visibly pushdown automaton $\mathcal{A}$ accepting the derivation trees of G, and another one $\mathcal{A}_E$ whose yield is the language of E. Furthermore, $L(\mathcal{A}_E)$ and $L(\mathcal{A})$ can be described by non recursive DTDs.

Let D be the domain that consists of trees with root r, and a unique subtree either in $L(\mathcal{A}_E)$ or in $L(\mathcal{A})$ below the root. Let Q_1, Q_2 denote two queries over D satisfying respectively (1) Q_1 selects all the leaves of t if t

consists of a root r and a subtree in $L(\mathcal{A})$ (then $\mathcal{V}iew\,(Q_1, t)$ represents the word w_G), or selects nothing but the root r if t consists of a root r and a subtree in $L(\mathcal{A}_E)$, and (2) Q_2 selects the leaves of every tree. Then $Q_1 \preccurlyeq_2 Q_2$ iff w_G does not belong to the language of E. This concludes the proof. □

We can conclude from Proposition 4.22 and Lemma 4.25 that

Theorem 4.26. *Comparison $\preccurlyeq_2$ is* PSPACE*-complete for query automata when the domain has bounded depth.*

Theorem 4.27. *Comparison $\preccurlyeq_2$ is* PSPACE*-complete for interval-bounded query automata.*

Theorem 4.28. *Comparison $\preccurlyeq_3$ is* PSPACE*-complete for query automata, when the domain has bounded depth.*

Proof. We have the hardness by using the same construction as in Lemma 4.25. Let us prove that this problem can be decided in polynomial space.

Here is a proof following a schema similar to $\preccurlyeq_2$: we define an automaton $\mathcal{A} = (\Sigma \times \Sigma_\varepsilon^2, S, \Gamma, I, F, R)$ with language $L(\mathcal{A}) = L_{Q_1 \otimes Q_2}$.

We transform $\mathcal{A}$ into a word transducer from $\mathcal{V}iew\,(Q_2, -)$ to $\mathcal{V}iew\,(Q_1, -)$. We build a word automaton $\mathcal{A}_w = (\Sigma \times \Sigma_\varepsilon^2, S \times \Gamma^k, I \times \{\varepsilon\}, F \times \{\varepsilon\}, R_w)$ equivalent to $\mathcal{A}$: for all $\eta \in \Sigma \times \Sigma_\varepsilon^2$, $u \in \Gamma^{\leqslant(k-1)}$, $q, q' \in S$, and all $\gamma \in \Gamma$, $\mathcal{A}_w$ has rule $(q, u) \xrightarrow{(op,\eta)} (q', u\gamma)$ iff $\mathcal{A}$ has rule $q \xrightarrow{(op,\eta):\gamma} q'$. $\mathcal{A}_w$ has rule $(q, u\gamma) \xrightarrow{(cl,\eta)} (q', u)$ iff $\mathcal{A}$ has rule $q \xrightarrow{(cl,\eta):\gamma} q'$.

From $\mathcal{A}_w$ we build automaton $\mathcal{B}_w = (\Sigma \times \Sigma_\varepsilon^2, S \times \Gamma^k, I \times \{\varepsilon\}, F \times \{\varepsilon\}, R_B)$, such that for all $x_1, x_2 \in \Sigma_\varepsilon$, $u \in \Gamma^{\leqslant(k-1)}$, $q, q' \in S$, and all $\gamma \in \Gamma$, $\mathcal{B}_w$ has rule $(q, u) \xrightarrow{(op,x_1,x_2)} (q', u\gamma)$ iff $x_1 \in \Sigma$ or $x_2 \in \Sigma$ and there exists $b \in \Sigma$ such that $\mathcal{A}_w$ has rule $(q, u) \xrightarrow{(op,(b,x_1,x_2))} (q', u\gamma)$. $\mathcal{B}_w$ has rule $(q, u) \xrightarrow{\varepsilon} (q', u\gamma)$ iff there exists $b \in \Sigma$ such that $\mathcal{A}_w$ has rule $(q, u) \xrightarrow{(op,(b,\varepsilon,\varepsilon))} (q', u\gamma)$. We add similar rules for the closing tags. We remark that the number of consecutive ϵ-transitions in a minimal (accepting) run of $\mathcal{B}_w$ over some input is bounded by $|\mathcal{A}_w|^k$.

Now, we can see $\mathcal{B}_w$ as a word transducer of polynomial size (remember that k is a fixed constant), and $Q_1 \preccurlyeq_2 Q_2$ if and only if that transducer is functional. We use the algorithm from [GI81, GI83] that decides functionality of word transducers in NLogspace. They use a result on the emptiness of automata with reversal-bounded counters to prove that whenever there is an input on which a word transducer T can produce two different outputs then there is such an input of size polynomial in T. Here, $\mathcal{B}_w$ is of exponential size, so that we cannot afford to build it, but we can simulate its transitions on-the-fly, and check for every input v of size polynomial in $|\mathcal{B}_w|$ – i.e., for every input of exponential size – if $\mathcal{B}_w$ can produce two different outputs on v. This gives a non-deterministic algorithm in polynomial space: guess the size of the input, and simulate $\mathcal{B}_w$ on-the-fly on this input. The result then follows from Savitch's theorem.

Theorem 4.29. *Comparison $\preccurlyeq_3$ is in* Exptime *for interval-bounded query automata.*

Proof. See the appendix, page 279. □

4.2.5. From MSO Queries to Views that Relabel Nodes

The previous results dealt with queries, i.e., views that do not relabel nodes. Comparison $\preccurlyeq_3$ can be used for views as well (without any need for adapting the definition). The results obtained carry over to views that relabel nodes. Similarly, the definition of comparison $\preccurlyeq_2$ can be adapted in a straightforward manner to deal with views that relabel nodes: for views V_1 and V_2, the definition becomes: $V_1 \preccurlyeq_2 V_2$ if

$$\forall t, t' \in D.\mathcal{V}iew\,(V_2, t) = \mathcal{V}iew\,(V_2, t') \implies \mathcal{V}iew\,(V_1, t) = \mathcal{V}iew\,(V_1, t')$$

Again, the results obtained for queries carry over to views that relabel nodes. The definition of $\preccurlyeq_1$ requires more thorough transformation: a possible definition would be: $V_1 \preccurlyeq_1 V_2$ if for every t the two following conditions are satisfied: (1)$N_{\mathcal{V}iew(V_1,t)} \subseteq N_{\mathcal{V}iew(V_2,t)}$ and (2) for every $n \in N_{\mathcal{V}iew(V_1,t)}$, $lab_{\mathcal{V}iew(V_1,t)}(n) = lab_{\mathcal{V}iew(V_2,t)}(n)$. With this definition, the polynomial time complexity for the first comparison still holds. But condition (2) may seem too restrictive. So it is not clear what is the natural notion of inclusion for views that relabel nodes.

4.2.6. Comparing $\mathcal{X}Reg$ Policies

Containment for $\mathcal{X}Reg$ Queries over a Non-recursive DTD We prove the Pspace-completeness of satisfiability for $\mathcal{X}Reg$ over non-recursive DTDs. This immediately gives the Pspace-completeness of the first comparison over non-recursive DTDs, as query containment and satisfiability are equivalent problems for $\mathcal{X}Reg$ according to Remark 3.5. Actually we claim that, given a non-recursive DTD D representing trees of maximal depth k, any Regular XPath formula ϕ encoding D has size $\Omega(k)$. This result can be proved as follows: let t be a tree of depth k in $L(D)$ then let $a_1, \ldots, a_k$ be the label of the nodes on the path from $root_t$ to some leaf of t with depth k. Necessarily, those labels are all distinct. If there are $1 \leqslant i < j \leqslant k$ such that ϕ does not explicitly contain the letters a_i and a_j, then the tree obtained from t by inverting labels a_i and a_j still satisfies ϕ, but does not belong to $L(D)$, which concludes the proof of the claim. As a consequence of this claim, we do not need to explicitly give the DTD as part of the input: satisfiability is in PSPACE and therefore PSPACE-complete as soon as the domain of the query is a non-recursive DTD. As a Corollary we obtain we obtain the same complexity for containment.

Theorem 4.30. *Satisfiability is* PSPACE*-complete for Regular XPath over non-recursive DTDs.*

Proposition 4.31. *Let $Q_{\mathcal{X}}$ and $Q_{\mathcal{X}'}$ be two root preserving $\mathcal{X}Reg$ queries. When the domain of $Q_{\mathcal{X}}$ is a non-recursive DTD, deciding $Q_{\mathcal{X}} \preccurlyeq_1 Q_{\mathcal{X}'}$ is* PSPACE*-complete.*

In order to prove membership in PSPACE it may be tempting to use a property of Regular XPath such as the small model property of PDL [BdRV01]. However, there exist (finitely) satisfiable $\mathcal{X}Reg$ formulae ϕ of size $O(n^2)$ whose smallest model has size 2^{2^n}, as exposed in [ABD+05]. We show in the appendix on page 277 that this gap can be improved using the technique from Kupferman and Rosenberg [KR10] presented for Theorem 3.23:

Remark 4.7. *There exist (finitely) satisfiable $\mathcal{X}Reg$ formulae ϕ of size $O(n)$ whose smallest model has size 2^{2^n}*

When the depth of all trees accepted by ϕ is bounded by $p(\phi)$ for some polynomial p, however, ϕ has a model of size at most exponential in ϕ, according to Corollary 3.14 and using the exponential conversion from $\mathcal{X}Reg$ formulae to NTAs.

Lemma 4.32. *There exists a polynomial p' such that for every $\mathcal{X}Reg$ formula ϕ of size n, if ϕ accepts only trees of depth at most $f(n)$ and $L(\phi) \neq \varnothing$ then $L(\phi)$ contains a tree of size $O(2^{p'(f(n))})$.*

Proof of Proposition 4.31. PSPACE-hardness is obvious since Regular XPath generalizes regular expressions, and containment for regular expressions is PSPACE-hard. With the small model property we have obtained we can sketch a PSPACE algorithm for satisfiability: we first guess the size of the tree satisfying ϕ. This tree has size exponential in ϕ by Lemma 4.32, so that its size can be represented using polynomial space only. We nondeterministically guess letter by letter the linearization of the tree, and the rule we apply. This rule can be verified in polynomial space according to Corollary 3.22. We only need to remember the stack of the automaton, which is of polynomial size by our hypothesis of a non-recursive DTD. Savitch's theorem allows to conclude. □

Determinacy for $\mathcal{X}Reg$ Queries over Non-recursive DTDs When the schema is a non-recursive DTD, we can prove that determinacy is in polynomial space with the same constructions as were used for satisfiability. Since *MSO* and $\mathcal{X}Reg$ have the same expressivity when the depth of the trees is bounded, $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$ if and only if $Q_1 \preccurlyeq_{2,MSO} Q_2$. So, by Proposition 4.19, $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$ if and only if $Q_1 \preccurlyeq_2 Q_2$.

Theorem 4.33. *Let Q_1 and Q_2 be two root preserving $\mathcal{X}Reg$ queries. When the domain of Q_1 is a non-recursive DTD, deciding $Q_1 \preccurlyeq_2 Q_2$ is* PSPACE*-complete.*

Proof. Let d denote the depth of the domain for query Q_2. To begin, we first check that $Q_1 \preccurlyeq_1 Q_2$, in polynomial space by Theorem 4.31. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ denote two automata over $\Sigma \times \Sigma_\varepsilon$ corresponding to queries Q_1 and Q_2. Using the construction in [CGLV09], for instance, we can assume that the size of $\mathcal{A}_1$ and $\mathcal{A}_2$ are at most exponential. Then, we use a pumping argument similar to Lemma 4.21: if there are two trees t, t' such that $Q_2(t) = Q_2(t')$ but $Q_1(t) \neq Q_1(t')$, then there are two such trees in which the number of children of every node is at most $p_0(|\mathcal{A}_1| + |\mathcal{A}_2|)^d$ for some polynomial p_0. Thus, there exists a polynomial p such that the number of children below each node in these trees is at most $2^{p(n)}$, where n is the sum of the size of Q_1 and Q_2. Since our hypothesis on the domain bounds the depth of the trees by a n, the size of t and t' is at most $2^{p(n)\times n}$. To sum up, we have proved that if there are two trees t, t' such that $Q_2(t) = Q_2(t')$ but $Q_1(t) \neq Q_1(t')$, then there are two such trees of size at most exponential in Q_1 and Q_2.

We cannot afford to build automata $\mathcal{A}_1$ and $\mathcal{A}_2$, but we simulate their execution on-the-fly: we guess the size of two trees t and t', which we can keep in memory as t and t' have exponential size. Then we check $Q_2(t) = Q_2(t')$ and $Q_1(t) \neq Q_1(t')$ by simulating the runs of $\mathcal{A}_1$ and $\mathcal{A}_2$ in polynomial space using Corollary 3.22.

Observe that we used the assumption bounding the depth of the domain in several assertions: in general query containment for $\mathcal{X}Reg$ is EXPTIME-complete, the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ cannot be simulated using polynomial space, and the expressive power of *MSO* and of $\mathcal{X}Reg$ are not the same. An assumption of that kind is thus necessary to avoid the difficulties observed in Proposition 4.37 for the more general interval-bounded setting.

We have already stated that $\preccurlyeq_{2,\mathcal{X}Reg}$ and $\preccurlyeq_{2,MSO}$ (and therefore $\preccurlyeq_2$) coincide over non-recursive DTDs because $\mathcal{X}Reg$ and *MSO* have the same expressive power in that case. The resulting query rewriting algorithm, however, is not very efficient: if we first compute an automaton rewriting Q_1 in terms of Q_2, we face a first exponential blowup. Converting the resulting query automaton into a $\mathcal{X}Reg$ query may involve another exponential blowup, so the whole construction is doubly exponential, and we do not have a matching lower bound.

Proposition 4.34. *Let Q_1 and Q_2 be two root preserving $\mathcal{X}Reg$ queries such that the domain of Q_1 is a non-recursive DTD and $Q_1 \preccurlyeq_2 Q_2$. We can compute in doubly exponential time $2^{2^{O(Q^3)}}$ a $\mathcal{X}Reg$ query Q satisfying* $\mathsf{Rewrite}(Q, Q_2) = Q_1$.

We obtain a result similar to Theorem 4.33 for comparison $\preccurlyeq_3$. It actually implies Theorem 4.33 by Propositions 4.15 and 4.31.

Proposition 4.35. *The problem of deciding $Q_1 \preccurlyeq_3 Q_2$ for $\mathcal{X}Reg$ queries Q_1 and Q_2 over non-recursive DTD D can be decided in polynomial space.*

Proof. We adapt the proof of Theorem 4.28. Once more, we use the translation from $\mathcal{X}Reg$ expressions into automata to build an automaton $\mathcal{A}$ of exponential size with language $L(\mathcal{A}) = L_{Q_1 \otimes Q_2}$. Actually, we do not build the automaton, because of its exponential size, but we simulate its transitions in polynomial space using Corollary 3.22 which also implies we can simulate in polynomial space the transitions of $\mathcal{B}_w$, where $\mathcal{B}_w$ is defined from $\mathcal{A}$ as in the proof of Theorem 4.28. The proof proceeds as for Theorem 4.28. □

Comparisons for Interval-bounded $\mathcal{X}Reg$ Queries For interval bounded $\mathcal{X}Reg$ queries, comparison $\preccurlyeq_{2,\mathcal{X}Reg}$ is not equivalent to $\preccurlyeq_2$. We prove that testing $\preccurlyeq_{2,\mathcal{X}Reg}$ can be reduced to $Memb(MSO, \mathcal{X}Reg)$, the membership problem, namely deciding *MSO* definability of $\mathcal{X}Reg$ formulae.

Proposition 4.36. *The problem of deciding $\preccurlyeq_{2,\mathcal{X}Reg}$ for interval bounded $\mathcal{X}Reg$ queries can be reduced in exponential time to $Memb(MSO, \mathcal{X}Reg)$.*

Proof. The reduction is immediate from the construction in Lemma 4.19 : we compute an automaton $\mathcal{A}$ with language $L_{Q_1 \otimes Q_2}$, test $Q_1 \preccurlyeq_2 Q_2$ and in this case the construction provides a query Q satisfying $\mathsf{Rewrite}(Q, Q_2) = Q_1$. All tests and the construction of Q require at most exponential time. Then, $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$ if and only if there exists a $\mathcal{X}Reg$ query equivalent to Q. □

However, since the exact complexity, or even the decidability of problem Memb(MSO,$\mathcal{X}Reg$) have not been established in the literature (to the best of our knowledge), this is of little help. Actually, the gap in expressiveness between *MSO* and $\mathcal{X}Reg$ has been established very recently [tCS08]. Thus, the following result sheds a new light on the problem of deciding $\preccurlyeq_{2,\mathcal{X}Reg}$.

Proposition 4.37. *$Memb(MSO, \mathcal{X}Reg)$ can be reduced in polynomial time to $\preccurlyeq_{2,\mathcal{X}Reg}$ with interval bounded $\mathcal{X}Reg$ annotations.*

Proof. Fix $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, R)$ a VPA, which we assume w.l.o.g. to be complete. That is, we assume $\mathcal{A}$ has a run (not necessarily accepting, of course) over all trees t in T_Σ. We build a DTD D and interval-bounded queries Q_1, Q_2 defined by $\mathcal{X}Reg$ expressions, such that $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$ iff there exists a $\mathcal{X}Reg$ filter f such that for every tree t, $(t, root_t) \vDash f$ if and only if $t \in L(\mathcal{A})$. We assume without loss of generality that $\Sigma \cap Q = \varnothing$. We build a DTD D over alphabet $\Sigma \cup Q$ defined via the following rules. Abusing notations for regular expressions, we use sets and denote by S the expression $s_1 \mid s_2 \mid \ldots \mid s_n$, for a set S consisting of elements $s_1, \ldots s_n$. For all $a \in \Sigma$, the production from a is defined by $a \to (Q \cdot \Sigma)^* \cdot Q$.

The proof works as follows: under r, D simulates a run of automaton $\mathcal{A}$ over a tree. Q_1 checks the simulation of the transitions and, when the run is valid and leads to an accepting state, Q_1 selects all nodes from the tree with label in Σ. A contrario, if either the run leads to rejection, or if the elements labeled in Σ simulate no valid run, Q_1 selects only the root. Q_2 selects all nodes from the tree with label in Σ when the run is valid, whether it is accepting or it leads to rejection, but selects only the root if the elements labeled in Σ simulate no valid run. The crux of the proof is to make sure with nodes labeled in Q that $\mathcal{V}iew\,(Q_1, D) = L(\mathcal{A})$, while $\mathcal{V}iew\,(Q_2, D)$ is the set of all trees over Σ.

This result is obtained with the following queries: let E be the set of all $(q_1, q_1', q_2, q_2', a)$ in $Q^4 \times \Sigma$ such that there exists some γ in Γ that verifies simultaneously $q_1 \xrightarrow{(op,a):\gamma} q_1'$ and $q_2' \xrightarrow{(cl,a):\gamma} q_2$. We define auxiliary $\mathcal{X}Reg$ filters: $f_\Sigma = \bigvee_{b \in \Sigma} \mathsf{self}::b$

$$f_{root} = \left(\bigvee_{q_i \in I} [\Downarrow[\,\mathsf{not} \Leftarrow]]/\mathsf{self}::q_i \right) \;\mathsf{and}\; \left(\bigvee_{q_f \in F} [\Downarrow[\,\mathsf{not} \Rightarrow]]/\mathsf{self}::q_f \right)$$

$$f^{q_1,q_2}_{q_1',q_2'} = (\Leftarrow::q_1) \;\mathsf{and}\; (\Rightarrow::q_2) \;\mathsf{and}\; (\Downarrow[\,\mathsf{not} \Leftarrow]/\mathsf{self}::q_1') \;\mathsf{and}\; (\Downarrow[\,\mathsf{not} \Rightarrow]/\mathsf{self}::q_2')$$

$$f_{valid} = \left[\mathsf{not} \left(\Downarrow^*/ \left[f_\Sigma \;\mathsf{and}\; \left(\mathsf{not} \bigvee_{(q_1,q_1',q_2,q_2',a) \in E} (\mathsf{self}::a) \;\mathsf{and}\; f^{q_1,q_2}_{q_1',q_2'} \right) \right] \right] \right) \right]$$

The two $\mathcal{X}Reg$ queries are defined as $Q_2 = [f_{valid}]/\Downarrow^*/[f_\Sigma] \cup \mathsf{self}[\,\mathsf{not} \Uparrow]$ and $Q_1 = [f_{valid} \;\mathsf{and}\; f_{root}]/\Downarrow^*/[f_\Sigma] \cup \mathsf{self}[\,\mathsf{not} \Uparrow]$. It should be clear that $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$ if and only if there exists a $\mathcal{X}Reg$ filter f such that for every tree t, $(t, root_t) \vDash f$ if and only if $t \in L(\mathcal{A})$. Actually, the two queries Q_1 and Q_2 are even upward-closed. □

From this proof and the expressivity gap between *MSO* and $\mathcal{X}Reg$ [tCS08], we can deduce that even for upward closed queries, $Q_1 \preccurlyeq_{2,MSO} Q_2$ does not imply $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$. Furthermore, in terms of expressivity, the queries Q_1 and Q_2 used in the proof belong to a small fragment of $\mathcal{X}Reg$ in that they do not use the full expressivity of the Kleene star. When the depth of the domain is not bounded, given any fragment $\mathcal{C}$ of $\mathcal{X}Reg$ and queries $Q'_1, Q'_2 \in \mathcal{C}$, $Q'_1 \preccurlyeq_2 Q'_2$ does not imply $Q'_1 \preccurlyeq_{2,\mathcal{C}} Q'_2$ as soon as $\mathcal{C}$ is expressive enough to define Q_1 and Q_2.

Corollary 4.38. *Let Q_1 and Q_2 be two upward-closed queries given by $\mathcal{X}Reg$ expressions, $Q_1 \preccurlyeq_2 Q_2$ needs not imply $Q_1 \preccurlyeq_{2,\mathcal{X}Reg} Q_2$.*

Because determinacy does not deal with expressiveness, we do not face the same difficulties related to the expressiveness of $\mathcal{X}Reg$ for Comparison $\preccurlyeq_3$ (for $\mathcal{X}Reg$):

Proposition 4.39. *Comparison $\preccurlyeq_3$ can be decided in exponential time for interval bounded $\mathcal{X}Reg$ queries.*

Proof. The proof first translates the $\mathcal{X}Reg$ expressions into automata, and proceeds as for Theorem 4.29: even if the automata have exponential size, the overall complexity remains exponential. □

As a corollary of Proposition 4.39 and 4.35, we obtain the same complexity bounds for $\preccurlyeq_2$:

Proposition 4.40. *Let Q_1 and Q_2 two Regular XPath queries. One can decide if $Q_1 \preccurlyeq_2 Q_2$ in exponential time for interval bounded $\mathcal{X}Reg$ queries, and in polynomial space over a non-recursive DTD.*

4.2.7. Other XPath Dialects

We consider Regular XPath as a natural fragment for expressing policies. Nonetheless other XPath fragments deserve some attention. CoreXPath$(*, \approx)$ extends Regular XPath with equality of paths [tCL09]. The syntax of filters becomes: $f ::= \mathsf{lab}() = b \mid \chi \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\, f \mid f \;\mathsf{and}\; f \mid f \;\mathsf{or}\; f \mid p \approx p$ and the semantics of the new operator is given by: $[\![p_1 \approx p_2]\!]_t = \{n \in N_t \mid \exists m \in N_t.(n, m) \in [\![p_1]\!]_t \cap [\![p_2]\!]_t\}$. Note that this XPath dialect allows to express the fact that a path expression loops on a node via $p \approx \mathsf{self}::\mathsf{true}$. Actually, [tCL09] provides an alternative definition of this dialect in terms of path expressions with loop tests. The authors show that every CoreXPath$(*, \approx)$ expression can be converted in polynomial time into an equivalent two-way alternating automaton with parity acceptance condition. This allows them to test emptiness of a CoreXPath$(*, \approx)$ expression in EXPTIME, but it also implies that our results for Regular XPath carry over to CoreXPath$(*, \approx)$: Theorem 4.30, Proposition 4.31, Proposition 4.39 and Proposition 4.35 ... also hold for CoreXPath$(*, \approx)$.

	VPA			$\mathcal{X}Reg$		
Schema	non-rec	IB	gen	non-rec	IB	gen
$\preccurlyeq_1$	PTime	PTime	PTime	Pspace-c	Exptime-c	Exptime-c
$\preccurlyeq_2$	Pspace-c [1]	Pspace-c [2]	undec	Pspace-c	Exptime-c	undec
$\preccurlyeq_3$	Pspace-c [1]	Exptime Pspace-h	undec	Pspace-c	Exptime-c	undec

[a] When the depth of the DTD is bounded by a *fixed* integer k, this problem becomes polynomial.

[b] When the constant for interval boundedness is a *fixed* integer k, this problem becomes polynomial.

Figure 4.7.: Summing up complexity for the three comparisons

Conditional XPath expressions can be viewed as a subset of Regular XPath expressions. Consequently, all upper bounds for Regular XPath also hold for Conditional XPath. Nevertheless, we did not use Conditional XPath for two reasons: first it is not expressive enough to encode DTDs, and then it does not allow easily to compose queries. Figure 4.7 summarizes our results on the complexity of policy comparisons $\preccurlyeq_1$,$\preccurlyeq_2$ and $\preccurlyeq_3$.

4.3. Beyond Pairwise Comparison

Here we outline how the methods developed above can help the database administrator to assess how much information is disclosed by a policy. We first sketch possible generalizations of view comparison when multiple views are considered or when n-ary queries come into play, and then discuss additional properties that can be verified using certain answers.

4.3.1. Policy Comparison in Presence of Multiple Views

A user may be allowed to take several roles and thus combine several views to gather more information. Can our result be generalized to compare sets of views? The answer depends on how the user may combine its views. We identify two particular settings. If the user can superimpose its different views into a single tree, and a set of views $\{V_1, V_2, \ldots, V_k\} \subseteq T_{\Sigma \times \Sigma_\varepsilon}$ can be modelled as a single view $V \subseteq T_{\Sigma \times \Sigma'}$ with $\Sigma' = \Sigma_\varepsilon^k$: the $(i+1)^{\text{th}}$ component of the tree alignments in view V correspond to the second component in view V_i. In that setting the problem of comparing two sets of views is reduced to the problem of comparing two single views, so the decidability results established in this section still apply.

If the user has no access to the relations ($follow, \preccurlyeq$) between the nodes from its different views, then the problem of determinacy becomes undecidable even for very simple views. This can be proved with an immediate encoding of PCP:

Example 4.7. *Let $n \in \mathbb{N}$ and $u_1, \ldots, u_n, v_1, \ldots, v_n \in \Sigma^*$ a PCP instance. Let D the DTD over alphabet $\{r, u, v, \#1, \ldots, \#n\} \cup \Sigma$ with root r and production rule $r \to uv(u(u_1\#1 + \cdots + u_n\#n)^* + v(v_1\#1 + \cdots + v_n\#n)^*)$. Consider annotations $\mathsf{ann}_0, \mathsf{ann}_1, \mathsf{ann}_2$ defined as follows:*

$\mathsf{ann}_0(r, u) = [\Rightarrow^*::u]$
$\mathsf{ann}_0(r, v) = [\Rightarrow^*::v]$
$\mathsf{ann}_0(r, \alpha) = \mathsf{false}$ *for every* $\alpha \in \Sigma \cup \{\#1, \ldots, \#n\}$

$\mathsf{ann}_1(r, u) = \mathsf{ann}_1(r, v) = [\,\mathsf{not}[\Leftarrow/\Leftarrow]]$
$\mathsf{ann}_1(r, 1) = \cdots = \mathsf{ann}_1(r, n) = \mathsf{false}$
$\mathsf{ann}_1(r, a) = \mathsf{true}$ *for every* $a \in \Sigma$

$\mathsf{ann}_2(r, u) = \mathsf{ann}_1(r, v) = \mathsf{false}$
$\mathsf{ann}_2(r, 1) = \cdots = \mathsf{ann}_2(r, n) = \mathsf{true}$
$\mathsf{ann}_2(r, a) = \mathsf{false}$ *for every* $a \in \Sigma$

Essentially, view 1 selects the first two children, and then only the letters from Σ. View 2 selects the indices, and view 0 selects the first child if the word is built from the u_is, and the second child otherwise. One can determine view 0 from the combination of views 1 and 2 if and only if there is no match for the instance of PCP encoded. This would not hold in the first setting because the position of the indices $\#i$ separating the u_is would then allow to distinguish whether the sequence is built from the u_is or the v_is.

4.3.2. Beyond Monadic Queries: n-ary Queries

When considering *MSO* n-ary queries ($n \geqslant 1$), there is no obvious notion of view tree: the representation via the maximal language, in particular, cannot be used. Thus, the notion of query composition needs another definition. The same question arises with determinacy: if we only consider queries returning tuples of node identifiers, then one may be unable to recover the structure of the trees, in particular in the case $n = 1$ the setting is different from the one investigated in this paper. For $n \geqslant 3$, modulo technical details we can encode the ancestor and next-sibling relations in the tuples returned by the query.

For any tree t, one possible solution is to define the view tree of t by an n-ary query Q as a pair $\mathcal{V}iew\,(Q, t) = (t_v, S)$ with $S \subseteq N_t^n$ the set of all tuples selected by Q in t and t_v the tree obtained from t by selecting every node that appears in at least one tuple of S, plus the root, the structure being inherited from t.

We represent every query Q as a language of alignments L_Q such that a tree t over $\Sigma_{n\text{-}ary} = \Sigma \times \{\varepsilon\} \times \{\varepsilon\} \cup \{(a, a, \alpha) \mid a \in \Sigma, \alpha \subseteq \{1, \ldots, n\}\}$ belongs to L_Q iff there exist a set $S \subseteq N_t$ and an n-uple $\mathbf{v} \in S$ satisfying the following two conditions: (1) $\mathcal{V}iew\,(Q, \pi_1(t)) = (\pi_2(t), S)$, (2) for every node x of t, $\pi_3(lab_t(x))$ is the set of all components of $\mathbf{v}$ that equal x.

Remark 4.8. *We observe that in any tree from L_Q and for every $i \leqslant m$ there exists at most one node x that contains i in its third component.*

A regular query Q will be represented by an automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L_Q$. We observe that while the second component helps to represent the monadic case as a restriction of this n-ary framework, we only consider queries in the n-ary case, and queries do not relabel nodes, so that the second component of each label is either equal to the first component, or is ε. In the case $n = 1$, then $S = N_{\pi_2(t)}$ in condition (1), and $\mathcal{V}iew\,(Q, t) = (\pi_2(t), S)$ can therefore be identified with $\pi_2(t)$.

Essentially, the second component displays all the nodes that are selected in at least one tuple, whereas the third component encodes one particular tuple that is selected. Several trees in $L(\mathcal{A})$ may therefore have the same first (and even first two) component(s) if Q selects several tuples in a tree.

We now extend the definition of determinacy to n-ary queries. Let Q_1 and Q_2 denote two n-ary queries over domain D. We say that Q_2 *determines* Q_1 and write $Q_1 \preccurlyeq_2 Q_2$ iff for every trees $t, t' \in D$, $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ implies $\mathcal{V}iew\,(Q_1, t) = \mathcal{V}iew\,(Q_1, t')$. This definition clearly extends the definition for the monadic case investigated in this dissertation, and as a result, remains undecidable in general. We shall prove that it remains decidable when D has bounded depth.

We generalize the definition of interval boundedness to n-ary queries, and qualify an n-ary query Q as k-interval bounded iff the set of alignments $\pi_{1,2}(L_Q)$ is k-interval bounded. Determinacy can still be decided for interval bounded queries using a pumping argument similar to Lemma 4.21.

Lemma 4.41. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two query automata expressing n-ary queries Q_1 and Q_2, such that Q_2 is interval bounded. If there are two trees t, t' such that $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ but $Q_1(t) \neq Q_1(t')$, then there are two such trees of depth exponential in the size of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. For any tree t and query Q_2, as we consider n-ary queries, we underscore that L_{Q_2} may contain several trees with first component t, so there is no practical way to represent $\mathcal{V}iew\,(Q_2, t)$ as a "decoration" of tree t. Instead, we define $t \otimes Q_2$ as the *set* of all trees in L_{Q_2} whose first component is t. The trees in $t \otimes Q_2$ thus have the same first two components but may differ in their third component. We next explain how to represent succinctly all possible runs of $\mathcal{A}_2$ on all trees of $t \otimes Q_2$ as a decoration of t. This decoration is not meant as a comprehensive representation of $Q_2(t)$, but only serves as a tool to decide whether $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ given two different trees t and t'.

Let $k, n \geqslant 1$ two naturals, and Q_1 and Q_2 two n-ary queries, represented by VPAs $\mathcal{A}_1 = (\Sigma, S_1, \Gamma_1, I_1, F_1, \Delta_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, \Gamma_2, I_2, F_2, \Delta_2)$, such that Q_2 is k-interval bounded and $Q_1 \preccurlyeq_1 Q_2$. Let also (t, t') be a pair of trees such that the first component of $\mathcal{V}iew\,(Q_2, t)$ and $\mathcal{V}iew\,(Q_2, t')$ are equal, that is, the tuples selected by Q_2 in t and t' are not necessarily the sames, but the set of all nodes that appear in at least one tuple are the same for t and t', and additionally they share the same parent and sibling relations in the view for Q_2. In other words, $\pi_2(t \otimes Q_2) = \pi_2(t' \otimes Q_2)$.

We first decorate each node of t in bottom-up order. For every $a \in \Sigma$, $m \geqslant 0$, and any a-labeled node $x \in N_{\mathcal{V}iew(Q_2,t)}$ such that in $\mathcal{V}iew\,(Q_2, t)$ node x has children $x_1, x_2, \ldots, x_m$, we first define decoration $deco_0$ which is basically a powerset construction, and then $deco$ which essentially indicates whether every tuple selected by Q_2 in t is also selected in t'.

The decoration $deco_0^t(x)$ is the set of all triples $(q, (q_1, \ldots, q_m), \alpha)$ with $q, q_1, \ldots, q_m \in (S_2)^2$ and $\alpha \subseteq \{1, \ldots, n\}$ such that there exists $t_0 \in t \otimes Q_2$ satisfying the following two conditions: (1) $lab_{t_0}(x) = (a, a, \alpha)$ and (2) there exists a run $\rho^{\uparrow}$ of $\mathcal{A}_2$ over $t_{0\restriction x}$ such that $\rho^{\uparrow}(x) = q$ and $\rho^{\uparrow}(x_i) = q_i$ for every $i \leqslant m$. Note that $m = 0$ when x is a leaf.

The decoration $deco_t^{t'}(x)$ is the union of two sets.

- The first set consists of all pairs (q, Fail) such that there exist $\alpha \subseteq \{1, \ldots, n\}$ and $q_1, \ldots, q_m \in (S_2)^2$ satisfying the following condition (a) together with (at least) one of (*b*) or (*c*): (a) $(q, (q_1, \ldots, q_m), \alpha) \in deco_0^t(x)$, and (b) there exists $i \leqslant m$ such that $(q_i, \mathrm{Fail}) \in deco_t^{t'}(x_i)$ or (c) there exist $S_1, \ldots, S_m \subseteq (S_2)^2$ satisfying the following conditions: (i) $(q_i, S_i) \in deco_t^{t'}(x_i)$, (ii) for every $q', q'_1, \ldots, q'_m$ such that $q'_i \in S_i$ for all $i \leqslant m$, $(q', (q'_1, \ldots, q'_m), \alpha) \notin deco_0^{t'}(x)$.

- The second set consists of some pairs (q, S) with $q \in (S_2)^2$ such that $(q, \mathrm{Fail}) \notin deco_t^{t'}(x)$, and $S \subseteq (S_2)^2$, obtained as follows. For each $\alpha \subseteq \{1, \ldots, k\}$, $q_1, \ldots, q_m \in (S_2)^2$, and each $S_1, \ldots, S_x \subseteq (S_2)^2$, (q, S) belongs to $deco_t^{t'}(x)$ if and only if S is the set of all $q' \in (S_2)^2$ for which there exist some $q'_1, \ldots, q'_m$ such that all the following four conditions are satisfied: (1) $q'_i \in S_i$ for all $i \leqslant m$, (2) $(q_i, S_i) \in deco_t^{t'}(x_i)$ for all $i \leqslant m$, (3) $(q, (q_1, \ldots, q_m), \alpha) \in deco_0^t(x)$, and (4) $(q', (q'_1, \ldots, q'_m), \alpha) \in deco_0^{t'}(x)$.

Claim: *We have $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ if and only if there is no pair from $(I_2 \times F_2) \times \{Fail\}$ in $deco_t^{t'}(root_t)$ and in $deco_{t'}^{t}(root_{t'})$.*

The claim can be proved using the following two invariants, which essentially state that for every $s, s' \in S_2$, $deco_t^{t'}(x)$ contains $((s, s'), \mathrm{Fail})$ if and only if $\mathcal{A}_2$ admits a run from s to s' on the subtree below x in t that "pre-selects" some tuple, whereas no run of $\mathcal{A}_2$ on the subtree below x in t' can "pre-select" the same tuple. We use the term of "pre-selection" because the tuple mentioned needs not be selected by Q_2 in t, though it might be (possibly after completing some components of the tuple with nodes that are not descendants of x).

Invariant: *For every $q \in (S_2)^2$ and every pair of trees t, t' over Σ such that $\pi_1(\mathcal{V}iew\,(Q_2, t)) = \pi_1(\mathcal{V}iew\,(Q_2, t'))$, if we denote by x the (common) root node of t and t', $(q, Fail) \in deco_t^{t'}(x)$ if and only if there exists a tree t_0 over $\Sigma_{n\text{-}ary}$ satisfying the following four conditions:*

1. $\pi_1(t_0) = t$,
2. $x \in N_{\pi_2(t_0)}$,
3. $\mathcal{A}_2$ *admits a run over* t_0 *with* $\rho^{\uparrow}(x) = q$
4. *there exists no tree* t'_0 *over* $\Sigma_{n\text{-}ary}$ *satisfying the three conditions (i)* $\pi_1(t'_0) = t'$, *(ii)* $\mathcal{A}_2$ *admits a run over* t'_0, *and (iii)* $\pi_{2,3}(t_0) = \pi_{2,3}(t'_0)$.

We observe that together with (2), (iii) implies $x \in N_{\pi_2(t'_0)}$.

Invariant: *For every $q \in (S_2)^2, S \subseteq (S_2)^2$ and every pair of trees t, t' over Σ with a common root node x, $(q, S) \in deco_t^{t'}(x)$ if and only if $(q, Fail) \notin deco_t^{t'}(x)$ and there exist two trees t_0 and t'_0 over $\Sigma_{n\text{-}ary}$ satisfying :*

1. $\pi_1(t_0) = t$,
2. $\pi_1(t'_0) = t'$,
3. $x \in N_{\pi_2(t_0)}$,
4. $\pi_{2,3}(t_0) = \pi_{2,3}(t'_0)$,
5. $\mathcal{A}_2$ *admits a run over* t_0 *with* $\rho^{\uparrow}(x) = q$
6. S *is the set of states* q' *such that* $\mathcal{A}_2$ *admits a run over* t_0 *with* $\rho^{\uparrow}(x) = q'$

Using the claim, we move on to the proof of the lemma. Let t and t' be two trees of minimal size such that $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$ but $Q_1(t) \neq Q_1(t')$. We claim that the depth of t and t' can be bounded in terms of $|\mathcal{A}_1|$ and $|\mathcal{A}_2|$.

Claim: *the depth of t and t' as defined above is bounded by $k \times 2^{2^{O(|S_1|^2 + |S_2|^2)}}$.*

If $\pi_1(\mathcal{V}iew\,(Q_1, t)) \neq \pi_1(\mathcal{V}iew\,(Q_1, t'))$ then Lemma 4.21 applies hence the claim holds. Consequently we assume from now on that $\pi_1(\mathcal{V}iew\,(Q_1, t)) = \pi_1(\mathcal{V}iew\,(Q_1, t')$, that is: the nodes appearing in at least one tuple selected by Q_1 are the same for t and t', so that $Q_1(t)$ and $Q_1(t')$ only vary in the tuples they select, not the nodes they make visible.

We define from $\mathcal{A}_1$ an automaton $\mathcal{B}_0$ such that $L(\mathcal{B}_0)$ is the set of all trees over $\Sigma_{n\text{-}ary}$ that do not belong to $L(\mathcal{A}_1)$. According to Theorem 3.5, one can build such a VPA $\mathcal{B}_0$ with $2^{|S_1|^2}$ states. From $\mathcal{A}_1$ and $\mathcal{B}_0$, one can easily build a VPA $\mathcal{B}_1$ that represents query $\overline{Q_1} : t \mapsto \{\mathbf{v} \in (N_t)^n \mid \mathbf{v} \notin Q_1(t)\}$. Again, B_1 needs no more than $2^{O(|S_1|^2)}$ states.

As the roles of t and t' are symmetric, we can assume without loss of generality that $Q_1(t)$ contains a tuple $\mathbf{v}$ that does not belong to $Q_1(t')$. Let therefore t_0 and t'_0 be the two trees such that

- $\pi_{1,2,3}(t_0) \in t \otimes Q_2$,
- $\pi_{1,4,5}(t_0) \in t \otimes Q_1$,
- $\pi_{1,2,3}(t'_0) \in t' \otimes Q_2$,
- $\pi_{1,4,5}(t'_0) \in t' \otimes \overline{Q_1}$,
- $\pi_{2,3,5}(t_0) = \pi_{2,3,5}(t'_0)$, and
- for every node $x \in N_{t_0}$ and $i \leqslant n$, i belongs to $\pi_5(lab_{t_0}(x))$ if and only if x is the i^{th} component of $\mathbf{v}$.

Let ρ_2 (resp. ρ'_2) denote an accepting run of the automaton $\mathcal{A}_2$ on $\pi_{1,2,3}(t_0)$ (resp. on $\pi_{1,2,3}(t'_0)$). Let also ρ_1 denote an accepting run of the automaton $\mathcal{A}_1$ on $\pi_{1,4,3}(t_0)$, and let ρ'_1 denote an accepting run of the automaton $\mathcal{B}_1$ on $\pi_{1,4,3}(t'_0)$. We also denote by $(\rho_2)^\uparrow$, $(\rho'_1)^\uparrow$ ... the corresponding functions that map a node x to the pair of states assigned by the run to the automaton before reading the opening tag and after processing the closing tag of x, as detailed on page 62.

We decorate every node x in $Q_2(t)$ (therefore also in $Q_2(t')$) with the tuple $\rho(x) = (\rho_2(x), \rho'_2(x), \rho_1(x), \rho'_1(x), deco_t^{t'}(x), deco_{t'}^{t}(x))$. First, we observe that $deco_t^{t'}(x)$ may take at most $2^{2^{O(|S_2|^2)}}$ different values, and so $\rho(x)$ may take at most $2^{2^{O(|S_1|^2+|S_2|^2)}}$ different values. [4]

Assume there is some node in $\mathcal{V}iew\,(Q_2, t)$ at depth greater than $k \times 2^{2^{O(|S_2|^2+|S_2|^2)}}$ in t or t'. Then there are two distinct nodes $n^\uparrow, n^\downarrow$ in $\mathcal{V}iew\,(Q_2, t)$ such that $n^\uparrow$ is an ancestor of $n^\downarrow$ and $\rho(n^\uparrow) = \rho(n^\downarrow)$. We observe that every node that appears in $\mathbf{v}$ either is a descendant of $n^\downarrow$ or is not a descendant of $n^\uparrow$ in both t and t'. For t this is because $\pi_{1,4,5}(t_0)$ belongs to $L(\mathcal{A}_1)$ and $\rho_1(n^\uparrow) = \rho_1(n^\downarrow)$, hence by Lemma 3.11, the tree obtained by repeating the "part" of t_0 between $n^\uparrow$ and $n^\downarrow$ is also accepted by $\mathcal{A}_1$, but for any tree t' in $L(\mathcal{A}_1)$ and any $i \leqslant n$, i cannot appear in the label of two distinct nodes of t' according to Remark 4.8. Therefore the fifth component of $lab_{t_0}(x)$ is $\varnothing$ for every node x that is a descendant of $n^\uparrow$ but not of $n^\downarrow$ in t_0. A symmetric argument proves the same result for t'.

Consequently, the trees t_1 and t'_1 obtained from t and t' by substituting the subtree below $n^\uparrow$ with the subtree below $n^\downarrow$ still satisfy $\mathbf{v} \in Q_1(t_1) \backslash Q_1(t'_1)$,

[4] Actually, $\rho(x)$ may take $2^{(|S_1|^2 \times 2^{O(|S_2|^2)})}$ different values, but the whole construction is inefficient anyway.

a fortiori $Q_1(t_1) \neq Q_1(t_1')$. Furthermore, we prove that $\mathcal{V}iew\,(Q_2, t_1) = \mathcal{V}iew\,(Q_2, t_1')$. First, $\pi_1(\mathcal{V}iew\,(Q_2, t_1)) = \pi_1(\mathcal{V}iew\,(Q_2, t_1'))$ because we have $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$, $\rho_2(n^\uparrow) = \rho_2(n^\downarrow)$, and $\rho_2'(n^\uparrow) = \rho_2'(n^\downarrow)$. Then $deco_t^{t'}(n^\uparrow) = deco_t^{t'}(n^\downarrow)$ and similarly $deco_{t'}^{t}(n^\uparrow) = deco_{t'}^{t}(n^\downarrow)$. Moreover, given any node x in the view for Q_2, the decorations $deco(x)$ only depend on the values of those decorations and on the states reachable in the children y of x in the view for Q_2, so that $deco_{t_1}^{t_1'}(x)$ and $deco_t^{t'}(x)$ are identical for every node x that is descendant of $n^\downarrow$ or that is not descendant of $n^\uparrow$. In particular $deco_{t_1}^{t_1'}(root_{t_1}) = deco_t^{t'}(root_t) = deco_{t'}^{t}(root_{t'}) = deco_{t_1'}^{t_1}(root_{t_1'})$. This implies that $\mathcal{V}iew\,(Q_2, t) = \mathcal{V}iew\,(Q_2, t')$. Thus, the existence of a node in $\mathcal{V}iew\,(Q_2, t)$ at depth greater than $k \times 2^{2^{O(|S_2|^2+|S_2|^2)}}$ in t or t' contradicts the minimality of t and t', which concludes the proof of the claim, hence the Lemma. □

We could prove along the same lines an horizontal pumping argument to bound the number of children in terms of $|\mathcal{A}_1|$ and $|\mathcal{A}_2|$, which gives the decidability of comparison $\preccurlyeq_2$ for n-ary queries.

4.3.3. Verifying Security Properties of a View

Instead of comparing several policies, one may wish to check security properties of a single view. We only mention one approach, based on certain answers, that is closely related to our work.

Verifying if some Specific "Sensitive" Information is Disclosed Libkin et Sirangelo [LS10] propose another approach based on certain answers: the database administrator specifies a Boolean query Q representing a secret, and this secret is considered to be disclosed by view V if there exists some tree t for which $Certain_V\,(Q; \mathcal{V}iew\,(V, t)) = \mathsf{true}$. There are a few differences between our formalisms and those of [LS10]. In [LS10], the view and query are specified by a single run query automaton with selecting states, a model equivalent to our query automata using maximal alignments. Also, the certain answers are parameterized with a domain in [LS10], because the automaton specifying the view may accept trees beyond that domain: $Certain_V^D\,(Q; t')$ equals true if and only if every t in D such that $\mathcal{V}iew\,(V, t) = t'$ satisfies Q. The authors prove by reduction from CFG universality that one cannot in general decide if a secret is disclosed.

Proposition 4.42 ([LS10]). *Given a view V and boolean query Q, all defined by automata, it is undecidable if there exists some tree t such that $Certain_V^D\,(Q; \mathcal{V}iew\,(V, t)) = \mathsf{true}$.*

When the view is upward-closed, however, the authors prove that the problem becomes decidable. They propose an algorithm to build an automaton $\mathcal{A}^*$

such that $L(\mathcal{A}^*) = \{t' \mid Certain_V^D(Q; t') = \mathsf{false}\}$. This, of course, implies the possibility to decide if there is a tree for which the secret is disclosed, since for instance we can check if $L(\mathcal{A}^*)$ is equal to the set of all possible view trees.

Proposition 4.43 ([LS10]). *Let V be a view, Q a boolean query, and D a domain. Given automata $\mathcal{A}_V$ for V, $\mathcal{A}_D$ for D, and $\mathcal{A}_{\neg Q}$ such that $t \in L(\mathcal{A}) \iff t' \not\models Q$, one can compute in polynomial time an automaton $\mathcal{A}^*$ such that $L(\mathcal{A}^*) = \{t' \mid Certain_V^D(Q; t') = \mathsf{false}\}$.*

$\mathcal{A}^*$ is essentially built from the product of $\mathcal{A}_D$ and $\mathcal{A}_{\neg Q}$ with $\mathcal{A}_V$: the selecting states of $\mathcal{A}_V$ identify the nodes that belong to the view, and so an analog of Proposition 4.2 (for upward-closed automata with selecting states) allows to build an automaton for those trees belonging to D that do not satisfy Q. Our results on interval-bounded views allow to generalize this proposition to interval-bounded views that may relabel nodes, via a straightforward adaptation of the proof in [LS10]. The complexity becomes exponential in k but polynomial for a fixed k. Libkin and Sirangelo conclude their analysis with the application of these results for XPath views and queries. Given a DTD D and Conditional XPath queries V and Q[5] for the view and secret, respectively, assuming Q_v to be upward-closed, the authors explain how to build an automaton $\mathcal{A}^*$ such that $L(\mathcal{A}^*) = \{t' \mid Certain_V^D(Q; t') = \mathsf{false}\}$ using their algorithm translating Conditional XPath queries into automata. The construction has complexity polynomial in $|D| \times 2^{O(|V|+|Q|)}$. This can also be generalized to Regular XPath using the translation from [CGLV09]. On the whole, this kind of reasoning about whether a secret is disclosed or not is complementary to our policy comparison definition. The analysis of information disclosed in terms of certain answers allows the verification of precise properties, whereas our comparisons are very general and therefore quite restrictive. One weakness of the certain answers analysis for security properties, however, lies in its vulnerability to statistical inference: if "most" trees with view t' satisfy Q, but one single tree (having view t') does not, then $Certain_V(Q; \mathcal{V}iew(V, t)) = \mathsf{false}$ (and $Certain_V(\neg Q; \mathcal{V}iew(V, t)) = \mathsf{false}$), yet a malicious user can still infer from view t' that Q is "likely" to hold. Other privacy notions allow to take this kind of statistical deductions into account.

[5] The formulation of the result in [LS10] is slightly different.

5. The View Update Problem

Contents

The previous chapter describes our framework for non-materialized security views. Rewriting a query Q_v from the user into a query over the source document is relatively straightforward, as evidenced in Theorems 4.7 and 4.8: the rewritten query was specified unambiguously as the composition of Q_v with the view. Managing updates defined by the user on the view, however, is a much more demanding task. First, the arbitrary combination of insertions and deletions quickly leads to undecidability problems for transducers. And secondly, an update defined by the user on the view does not in general define unambiguously the update that should be applied on the source document. Therefore, choosing the right update on the source requires additional information, or an arbitrary choice function.

5.1. Formalization

Before we introduce formal definitions, let us illustrate with our software projects example the problems raised by updates on views. Suppose again we have a database containing projects of two kinds: stable projects, and projects under development. The process is controlled by two different authorities A_1, A_2 that certify the projects independently by attaching some certificate c_1 (resp. c_2) to the projects in the database. Every stable project possesses a certificate from each authority. Projects that are not yet certified remain under development, and once a project has received both certificates, it becomes stable. For the sake of clarity, we only keep the `name` nodes, and remove other informations such as `license`, etc. in the following specifications, so that the database schema is given by the following DTD:

$$\begin{array}{lcl}
\texttt{projects} & \rightarrow & \texttt{project}^* \\
\texttt{project} & \rightarrow & \texttt{name}, (\texttt{stable} \mid \texttt{dev}) \\
\texttt{dev} & \rightarrow & \texttt{c1}? \mid \texttt{c2}? \\
\texttt{stable} & \rightarrow & \texttt{c1}, \texttt{c2}
\end{array}$$

Since each authority should ignore the status of the project and work independently, authority A_1 gets only a view of the database, that hides certificates `c2` and renames both `dev` and `stable` elements with a more general `docs` label for documents. Authority A_1 should not even be aware that it has only access to a view of the database instead of the whole database. This means in particular that A_1 does not get DTD D and instead gets a schema for its view, that consists of the following three rules: $\texttt{projects} \rightarrow \texttt{project}^*$, $\texttt{project} \rightarrow \texttt{name}, \texttt{docs}$, and $\texttt{repository} \rightarrow \texttt{c1}?$

Now, authority A_1 may wish to delete all its certificates, via an XQUF query like $Q_V = \texttt{delete}\ \ \texttt{/projects/project/docs/c1}$. This update should not be applied directly on the database, since there are no `docs` elements in it. Besides, deleting `c1` element under a `stable` project would lower the status of this project from `'stable'` to `'dev'`. The update function Q_V should thus be first translated into some query like

```
delete /projects/project/dev/c1,
delete /projects/project/stable/c1,
for $p in /projects/project/stable return rename node $p as dev
```

This chapter focuses on such update translation problems.

5.1.1. Equivalence of Editing Scripts

We wish to emphasize that our notion of update takes node identifiers into account. Different editing scripts can define the same transformation between input and output document up to isomorphism, but we still wish to distinguish them. For instance, the three editing scripts $(r,r)((\varepsilon,b)(a,\varepsilon))$, $(r,r)((a,\varepsilon),(\varepsilon,b))$, and $(r,r)((a,b))$ define the same transformation from input tree $r(a)$ to output tree $r(b)$ when we consider those two trees up to isomorphism. However, we wish to distinguish the update performed by the two former scripts from the update performed by the latter script. Intuitively, the two first scripts insert a b-labeled node and delete an a-labeled node alike, save they do it in different order. The third script renames an a-labeled node as b. Therefore, the first two scripts are equivalent, and are different from the third one. More generally, two editing scripts are equivalent if we can obtain each of them from the other by (repeatedly) commuting a subtree labeled with insertions with an adjacent subtree labeled with deletions. This is formalized below.

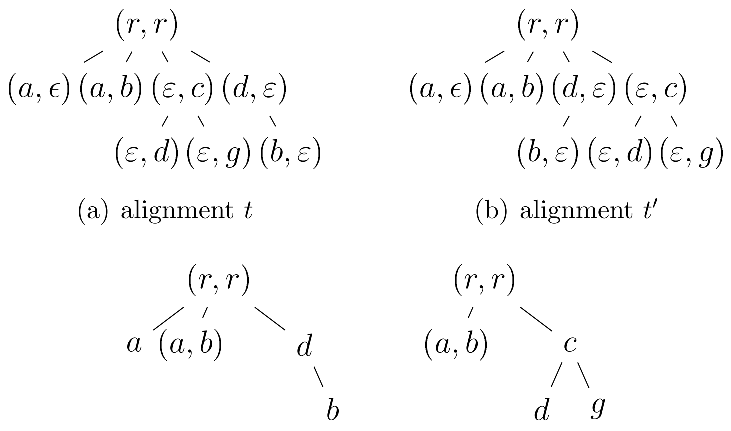
Consider the two morphisms $\Phi_1, \Phi_2 : \Sigma_{\text{edit}} \to \Sigma \cup \Sigma^2 \cup \{\varepsilon\}$ defined by:

$$\Phi_i(\alpha_1, \alpha_2) = \begin{cases} \alpha_i & \text{if } \alpha_{(3-i)} = \varepsilon \text{ or } \alpha_i = \varepsilon \\ (\alpha_1, \alpha_2) & \text{otherwise} \end{cases}$$

Definition 5.1. *Two editing scripts t and t' are* equivalent*, if $\Phi_1(t) = \Phi_1(t')$ and $\Phi_2(t) = \Phi_2(t')$. In this case we write $t \sim t'$, as $\sim$ is clearly an equivalence relation.*

Notation. *We define the equivalence class of an editing script t as $[t] = \{t' \mid t' \sim t\}$. We extend these definitions to sets of editing scripts: $[L] = \bigcup_{t \in L}[t]$, and $L \sim L'$ if $[L] = [L']$.*

Let us note that $\Phi_1(L) = \Phi_1(L')$ and $\Phi_2(L) = \Phi_2(L')$ does not imply $L \sim L'$. Figure 5.1 represents two editing scripts t and t', and their images by the morphisms Φ_1 and Φ_2 as a witness for $t \sim t'$.



(a) alignment t

(b) alignment t'

(c) $\Phi_1(t) = \Phi_1(t')$

(d) $\Phi_2(t) = \Phi_2(t')$

Figure 5.1.: Two equivalent trees t and t'.

As seen before, a set of editing scripts L induces a binary relation of input and output trees $\{(\pi_1(u), \pi_2(u)) \mid u \in L\}$. If two editing scripts are equivalent, they induce the same relation, but the converse is false in the general case. However, it is true when the scripts contain no insertion (resp. no deletion, resp. no renaming). Equivalence of two regular sets of editing scripts is undecidable; this can be easily deduced from undecidability of equivalence of two word transducers [Gri68] or undecidability results for trace languages [AH87]. Let us also note that even when L is a regular set of words, the set $[L]$ needs not even be context-free: consider the set of editing scripts $\{(r,r)(w) \mid w \in ((a,\varepsilon)(\varepsilon,b))^* \, ((c,\varepsilon)(\varepsilon,d))^*\}$.

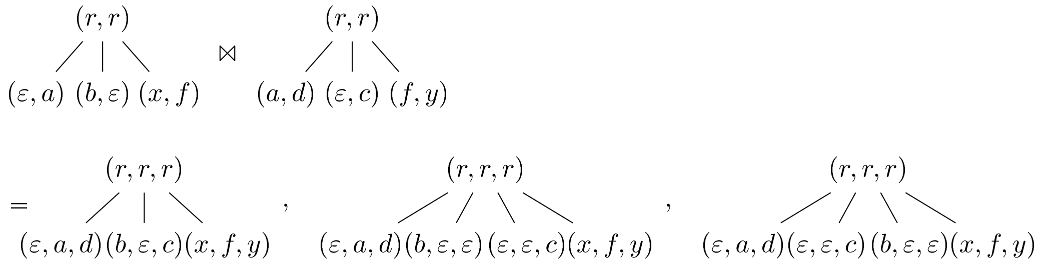

(r,r)
(ε,a) (b,ε) (x,f)
$\bowtie$
(r,r)
(a,d) (ε,c) (f,y)
$=$
(r,r,r)
$(\varepsilon,a,d)(b,\varepsilon,c)(x,f,y)$
,
(r,r,r)
$(\varepsilon,a,d)(b,\varepsilon,\varepsilon)(\varepsilon,\varepsilon,c)(x,f,y)$
,
(r,r,r)
$(\varepsilon,a,d)(\varepsilon,\varepsilon,c)(b,\varepsilon,\varepsilon)(x,f,y)$

Figure 5.2.: Synchronization of two editing scripts.

Remark 5.1. *If t, t' (resp. L, L') are editing scripts (resp. sets of editing scripts) over the alphabet $\Sigma \times \Sigma_\varepsilon$ or over the alphabet $\Sigma_\varepsilon \times \Sigma$, then equivalence coincides with equality. That is, $t \sim t'$ iff $t = t'$ (resp. $L \sim L'$ iff $L = L'$).*

The inverse of an editing script is an editing script having the same tree structure but in which labels are inverted, that is, (α, β) becomes (β, α), for $\alpha, \beta \in \Sigma_\varepsilon$. This can be achieved with the morphism $\pi_{2,1}$.

Definition 5.2. *For an editing script t, we denote by t^{-1} its inverse editing script defined by $t^{-1} = \pi_{2,1}(t)$. We extend this definition to sets of editing scripts: the inverse of a set L of editing scripts is $L^{-1} = \{s^{-1} \mid s \in L\}$.*

We point out that for any automaton $\mathcal{A}$ that accepts a set of editing scripts L, the automaton obtained from $\mathcal{A}$ by inverting the label in every transition accepts L^{-1}.

5.1.2. Composition of Editing Scripts

In order to define compositions of updates, we define synchronization of editing scripts. The definitions are given in terms of sets of editing scripts, but the definition for single editing scripts can be deduced by identifying an editing script with the singleton containing that script.

Definition 5.3. *For $n \geqslant 2$ sets of editing scripts L_1, L_2, ... L_n, their* synchronization *$L_1 \bowtie L_2 \bowtie ... \bowtie L_n$ is the set of trees t over $\Sigma_{edit,n+1}$ such that for all $1 \leqslant i \leqslant n$ and $\pi_{i,i+1}(t) \in L_i$.*

Figure 5.2 presents the synchronization of two editing scripts.

Remark 5.2. *Consider two sets of editing scripts L_1, L_2 and let $u \in L_1 \bowtie L_2$. Let $u_1 \in L_1$ and $u_2 \in L_2$ be the witnesses for $u \in L_1 \bowtie L_2$, that is, $\pi_{1,2}(u) = u_1$ and $\pi_{2,3}(u) = u_2$. Remark that in this case $u \in u_1 \bowtie u_2$. Then $\pi_2(u_1) = \pi_1(u_2)$, as both are equal to $\pi_2(u)$. This intuitively means that the synchronization of two editing scripts u_1, u_2 (resp. of two sets of*

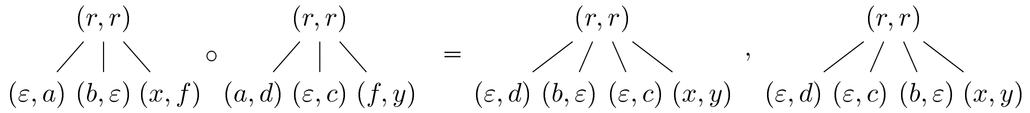

(r,r)
(ε,a) (b,ε) (x,f)
$\circ$
(r,r)
(a,d) (ε,c) (f,y)
$=$
(r,r)
(ε,d) (b,ε) (ε,c) (x,y)
,
(r,r)
(ε,d) (ε,c) (b,ε) (x,y)

Figure 5.3.: Composition of two editing scripts.

editing scripts L_1, L_2) is obtained by "gluing" the two trees (resp. the two sets of trees) around a common "middle" component $\pi_1(u_2) = \pi_2(u_1)$ (resp. $\pi_1(L_2) \cap \pi_2(L_1) \neq \varnothing$). This is actually where the term "synchronization" comes from.

Regularity is preserved by synchronization, as established in the following proposition.

Proposition 5.1. *Given regular sets of editing scripts L_1, L_2, ... L_n, their synchronization $L_1 \bowtie L_2 \bowtie ... \bowtie L_n$ is a regular set of alignments.*

Proof. By definition, $L_1 \bowtie L_2 \bowtie ... \bowtie L_n$ is obtained as the intersection of $(\pi^{n+1}_{1,2})^{-1}(L_1)$, $(\pi^{n+1}_{2,3})^{-1}(L_2)\dots$, and $(\pi^{n+1}_{n,n+1})^{-1}(L_n)$, and this set is regular by Proposition 3.3. □

The composition of two editing scripts is the set of editing scripts obtained by projecting their synchronization on its first and last component. We denote by $L_1 \circ L_2$ the composition of L_1 and L_2 with the intuitive semantics that editing operations from L_1 are applied first, then operations from L_2 are applied on the resulting document. Since composition is to represent the composition of editing operations, we forbid to "recover" a node once it has been deleted.

Definition 5.4. *Let L_1 and L_2 two sets of editing scripts. The* composition *of L_1* and *L_2 is defined as $L_1 \circ L_2 = \pi_{1,3}\left((L_1 \bowtie L_2) \cap L_{corr}\right)$, where L_{corr} is the set of all trees having no nodes labeled with a tag in $\Sigma \times \{\varepsilon\} \times \Sigma$.*

Example 5.1. *Let L_1 represent the transformation that relabels every a node into b in trees of T_Σ, i.e., L_1 is the set of trees over alphabet $\{(a,b)\} \cup \{(x,x) \mid x \in \Sigma\backslash\{a\}\}$. Similarly, let L_2 represent the transformation that relabels b nodes into c. Then $L_1 \circ L_2$ is the set of all trees over alphabet $\{(a,c),(b,c)\} \cup \{(x,x) \mid x \in \Sigma\backslash\{a,b\}\}$.*

As another example, Figure 5.3 represents the composition of the two editing scripts from Figure 5.2. The first of the three editing scripts in the synchronization on Figure 5.2 is removed before applying the projection because it contains a node labeled (b,ε,c).

Composition of editing scripts preserves regularity, and the corresponding automaton is constructed in polynomial (quadratic) time. The construction can be obtained as a minor modification of the one for Theorem 4.8, taking into account insertions and upward-closure of the tree alignments.

Proposition 5.2. *Operation $\circ$ is associative.*

Proof. Let L_{corr} denote the set of all trees over $\Sigma_{\mathrm{edit},3}$ that have no node labeled with a tag in $\Sigma \times \{\varepsilon\} \times \Sigma$, and $L_{\mathrm{corr},4}$ denote the set of all trees over $\Sigma_{\mathrm{edit},4}$ that have no node labeled with a tag in $\Sigma\times\{\varepsilon\}\times\Sigma\times\Sigma_\varepsilon$, $\Sigma_\varepsilon\times\Sigma\times\{\varepsilon\}\times\Sigma$, or $\Sigma \times \{\varepsilon\} \times \{\varepsilon\} \times \Sigma$.

First, we note that $\pi^3_{1,3}(u \bowtie v) \bowtie w = \pi^4_{1,3,4}(u \bowtie v \bowtie w)$. Let m in $(u{\circ}v){\circ}w$, i.e. in $\pi^3_{1,3}(\pi^3_{1,3}(u \bowtie v \cap L_{\mathrm{corr}}) \bowtie w \cap L_{\mathrm{corr}})$. Then, $m \in \pi^3_{1,3}(\pi^3_{1,3,4}(u \bowtie v \bowtie w \cap L_{\mathrm{corr},4}))$, so $m \in \pi^4_{1,4}(u \bowtie v \bowtie w{\cap}L_{\mathrm{corr},4})$. Let $m \in \pi^4_{1,4}\left((u \bowtie v \bowtie w) \cap L_{\mathrm{corr},4}\right)$, then clearly $m \in u \circ (v \circ w)$: there is $m_0 \in (u \bowtie v \bowtie w) \cap L_{\mathrm{corr},4}$ such that $m = \pi^4_{1,4}(m_0)$. Since $m_0 \in L_{\mathrm{corr},4}$, $\pi_{1,2,4}(m_0) \in L_{\mathrm{corr}}$, and $\pi_{1,2,3}(m_0) \in L_{\mathrm{corr}}$. Therefore, $\pi_{1,2,4}(m_0) \in (u \bowtie (v \circ w)) \cap L_{\mathrm{corr}}$. This implies $m = \pi_{1,4}(m_0) = u \circ (v \circ w)$. □

In particular, taking S to be the set of all sets of editing scripts, $(S, \circ)$ is a monoid, with neutral element the set of all editing scripts over $\{(a, a) \mid a \in \Sigma\}$.

Remark 5.3. *As we could expect, $(L_1 \circ L_2)^{-1} = L_2^{-1} \circ L_1^{-1}$ and the relation associated with L^{-1} is the inverse relation of the relation associated with L. However, $L \mapsto L^{-1}$ is not the inverse operation associated to the binary operation $\circ$. Indeed $(S, \circ)$ is not a group: not every set L has an inverse for operation $\circ$, whereas L^{-1} is always defined.*

We prove a first technical lemma regarding the synchronization of equivalent editing scripts. We denote by p_1 the morphism defined by : $p_1(\varepsilon, \alpha, \beta) = \varepsilon$ and $p_1(a, \alpha, \beta) = (a, \alpha, \beta)$ for all $a \in \Sigma$, $\alpha, \beta \in \Sigma_\varepsilon \times \Sigma_\varepsilon$.

Lemma 5.3. *For all editing scripts w, w' over $\Sigma_{\mathit{edit},3}\backslash\Sigma\times\{\varepsilon\}\times\Sigma$, if $\pi_{1,2}(w) \sim \pi_{1,2}(w')$ and $\pi_{2,3}(w) \sim \pi_{2,3}(w')$, then $p_1(w) = p_1(w')$.*

Proof. The definitions of equivalence for words is an immediate adaptation of Definition 5.1 since morphisms can be evaluated on words. The result for editing scripts follows using the linearization: $p_1(\mathit{lin}(w) = \mathit{lin}(p_1(w)$. We fix two words w, w' over $\Sigma_{\mathrm{edit},3}\backslash\Sigma \times \{\varepsilon\} \times \Sigma$ such that $\pi_{1,2}(w) \sim \pi_{1,2}(w')$ and $\pi_{2,3}(w) \sim \pi_{2,3}(w')$. For every word m, integer k and set $S \subseteq \Sigma_{\mathrm{edit},3}$, we denote by $m[k]$ the k^{th} letter of word m, and by $\mathit{Before}^m(k, S)$ the number of elements with label in S among $m[1], m[2], \ldots m[k-1]$. Clearly, for all $k \geqslant 1$, $a, b \in \Sigma, \gamma \in \Sigma_\varepsilon$:

$$\begin{array}{lcl} (p_1(w))\,[k] = (a,\varepsilon,\varepsilon) & \text{iff} & (\Phi_1(\pi_{1,2}(w)))\,[k] = (a,\varepsilon) \\ (p_1(w))\,[k] = (a,b,\gamma) & \text{iff} & \left\{ \begin{array}{l} (\Phi_1(\pi_{1,2}(w)))\,[k] = (a,b) \\ \text{and } (\Phi_1(\pi_{2,3}(w)))\,[k^w_1] = (b,\gamma) \end{array} \right. \end{array}$$

where k_1^w is the number defined in the following equation:
$k_1^w = k + \mathit{Before}^w\left(k, \{\varepsilon\} \times \mathit{\Sigma} \times \mathit{\Sigma}_\varepsilon\right) - \mathit{Before}^w\left(k, \mathit{\Sigma} \times \{\varepsilon\} \times \{\varepsilon\}\right)$. Given the definition of p_1, $(p_1(w))[k]$ is necessarily of the form $(a, \varepsilon, \varepsilon)$ or of the form (a, b, γ) for some $a, b \in \Sigma$, $\gamma \in \Sigma_\varepsilon$. Similarly,

$$\begin{array}{lll} (p_1(w'))[k] = (a, \varepsilon, \varepsilon) & \text{iff} & (\Phi_1(\pi_{1,2}(w')))[k] = (a, \varepsilon) \\ (p_1(w'))[k] = (a, b, \gamma) & \text{iff} & \begin{cases} (\Phi_1(\pi_{1,2}(w')))[k] = (a, b) \\ \text{and } (\Phi_1(\pi_{2,3}(w')))[k_1^{w'}] = (b, \gamma) \end{cases} \end{array}$$

From these equations we conclude that $(p_1(w))[k] = (a, \varepsilon, \varepsilon)$ if and only if $(p_1(w'))[k] = (a, \varepsilon, \varepsilon)$. Furthermore, we observe that if $(p_1(w))[k] = (a, b, \gamma)$, then there exists $\gamma' \in \Sigma_\varepsilon$ such that $(p_1(w'))[k] = (a, b, \gamma')$, since by hypothesis $\Phi_1(\pi_{1,2}(w)) = \Phi_1(\pi_{1,2}(w'))$. For such a k, this implies

$$\mathit{Before}^w\left(k, \{\varepsilon\} \times \mathit{\Sigma} \times \mathit{\Sigma}_\varepsilon\right) = \mathit{Before}^{w'}\left(k, \{\varepsilon\} \times \mathit{\Sigma} \times \mathit{\Sigma}_\varepsilon\right)$$

since by hypothesis $\Phi_2(\pi_{1,2}(w)) = \Phi_2(\pi_{1,2}(w'))$, and

$$\mathit{Before}^w\left(k, \mathit{\Sigma} \times \{\varepsilon\} \times \{\varepsilon\}\right) = \mathit{Before}^{w'}\left(k, \mathit{\Sigma} \times \{\varepsilon\} \times \{\varepsilon\}\right)$$

since by hypothesis $\Phi_1(\pi_{1,2}(w)) = \Phi_1(\pi_{1,2}(w'))$. Therefore, $k_1^w = k_1^{w'}$, so that $(p_1(w))[k] = (p_1(w'))[k]$. This concludes the proof. □

As a corollary of this lemma, for every editing scripts t and t' over $\Sigma_{\text{edit},3} \backslash \Sigma \times \{\varepsilon\} \times \Sigma$, if $\pi_{1,2}(t) \sim \pi_{1,2}(t')$ and $\pi_{2,3}(t) \sim \pi_{2,3}(t')$, then $\Phi_1(\pi_{1,3}(t)) = \Phi_1(\pi_{1,3}(t'))$. By symmetry, we get $\Phi_2(\pi_{1,3}(t)) = \Phi_2(\pi_{1,3}(t'))$, and this proves both Propositions 5.4 and 5.5 below. Propositions 5.4 states that, although the composition of two editing scripts may result in several editing scripts, those editing scripts are equivalent. Propositions 5.5 states that our equivalence relation is stable under composition.

Proposition 5.4. *Given editing scripts u and w, all editing scripts in $u \circ w$ are equivalent.*

Proposition 5.5. *Given editing scripts u, u', w, w', if $u \sim u'$ and $w \sim w'$, then $u \circ w \sim u' \circ w'$.*

Note that neither Proposition 5.4 nor Proposition 5.5 would hold if we did not intersect $\pi_{1,3}\left(L_1 \bowtie L_2\right)$ with L_{corr} in the definition of composition. For instance, let u, u' and v denote the three following editing scripts of depth one: $u = (r, r)((b, \varepsilon), (\varepsilon, a))$, $u' = (r, r)((\varepsilon, a), (b, \varepsilon))$, and $v = (r, r)((a, d), (\varepsilon, c))$. Clearly, $u \sim u'$, but $\pi_{1,3}(u \bowtie v)$, which contains the single editing script $(r, r)((b, \varepsilon), (\varepsilon, d), (\varepsilon, c))$, is not equivalent to $\pi_{1,3}(u' \bowtie v)$ which contains editing scripts $(r, r)((\varepsilon, d), (b, \varepsilon), (\varepsilon, c))$ and $(r, r)((\varepsilon, d), (\varepsilon, c), (b, \varepsilon))$, but also $(r, r)((\varepsilon, d), (b, c))$. We observe also that the second and third editing scripts in $\pi_{1,3}(u' \bowtie v)$ are not equivalent.

One could wonder if the converse of Proposition 5.5 is true, namely: given two editing scripts u and w, can we obtain every editing script equivalent to $u \circ w$ as the composition of two editing scripts u' and w' respectively equivalent to u and w? Unfortunately, this property is not true in general, as illustrated by Example 5.2.

Example 5.2. *Let $u = (r, r)((\varepsilon, a), (b, b))$ and $w = (r, r)((a, a), (b, \varepsilon))$. The unique editing script in $u \circ w$ is $(r, r)((\varepsilon, a), (b, \varepsilon))$, but there are no editing scripts u' and w' respectively equivalent to u and w such that $u \circ w$ contains $(r, r)((b, \varepsilon), (\varepsilon, a))$ since u and w are the unique elements in their respective equivalence class.*

In the following, we denote by u_s and u_v editing scripts with the intended meaning that u_v should be applied on the view and u_s on the source. We also denote by V a view, and by L a set of editing scripts.

Properties of Views for Composition Views have the following noteworthy properties with respect to composition and equivalence:

Lemma 5.6. *For all editing scripts u_s, u'_s and u_v, for all view V*

1. *$u_s \bowtie V$, $V^{-1} \bowtie u_s$, and $V^{-1} \bowtie u_s \bowtie V$ are singletons or empty.*
2. *$[V \circ u_v] = V \circ [u_v]$.*

Proof. 1. By definition of synchronization and by Remark 5.2, (a) $u \in u_s \bowtie V$ only if $\exists u_2 \in V$ s.t. $\pi_1(u_2) = \pi_2(u_s)$ and $u \in u_s \bowtie u_2$. Now, by definition of views and using Remark 5.1, there exists at most one editing script u_2 in V such that $\pi_1(u_2) = \pi_2(u_s)$ and, thus using (a), we have (b) $u_s \bowtie V = u_s \bowtie u_2$. Another observation is that (c) for all editing scripts s, s', if s' is over the alphabet $\Sigma \times \Sigma_\varepsilon$, then $s \bowtie s'$ consists of a single tree (having the same structure as s), or is empty whenever $\pi_2(s) \neq \pi_1(s')$. From (b) and (c) we deduce that $u_s \bowtie V$ is a singleton or empty. Using similar arguments and Remark 5.3, we can show that $V^{-1} \bowtie u_s$ and $V^{-1} \bowtie u_s \bowtie V$ also have cardinality at most one.

2. Inclusion $[V \circ u_v] \supseteq V \circ [u_v]$ follows from Proposition 5.5. We give a proof for $[V \circ u_v] \subseteq V \circ [u_v]$ in the case of words. This proof can be extended to trees like the one for Proposition 5.5 by considering the linearization. Fix $v \in V$, u, and $\mathbf{w} \in (v \bowtie u) \cap L_{\text{corr}}$. Let w' a word over Σ_{edit} such that $w' \sim \pi^3_{1,3}(\mathbf{w})$. Then $|w'| = |\mathbf{w}|$ because $\mathbf{w}$ contains no letter in $\{\varepsilon\} \times \Sigma \times \{\varepsilon\}$. We are going to build some $u' \sim u$ and $\mathbf{w}' \in (v \bowtie u') \cap L_{\text{corr}}$ such that $\pi^3_{1,3}(\mathbf{w}') = w'$. Intuitively, $w' \sim \pi^3_{1,3}(\mathbf{w})$ if and only if w' can be obtained from $\pi^3_{1,3}(\mathbf{w})$ by (repeatedly) commuting nodes labeled (a, ε) with adjacent nodes labeled (ε, b) for some pairs of

letters $a, b \in \Sigma$. A letter (a, ε) must correspond in $\mathbf{w}$ to a letter of the form $(a, \varepsilon, \varepsilon)$ or of the form (a, d, ε) for some $d \in \Sigma$, while a letter (ε, b) must correspond in $\mathbf{w}$ to a letter of the form $(\varepsilon, \varepsilon, b)$. This explains how u' can be built from u, by commuting (d, ε) with (ε, b) if necessary.

The following diagram considers a view $v = (a, \varepsilon)(c, d)(a, a)$ and an editing script $u = (d, \varepsilon)(\varepsilon, b)(a, c)$, with $\mathbf{w}$, w and w' has above. The diagram illustrates how an editing script $u' \sim u$ can be computed such that $w' \in v \circ u$ and also provides a corresponding $\mathbf{w}'$.

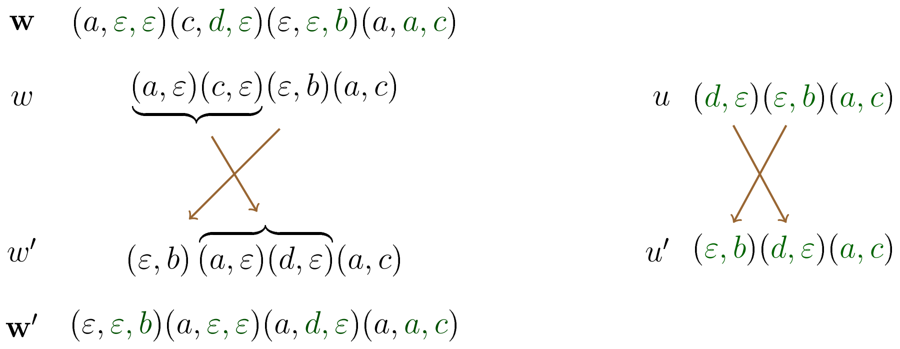

$\mathbf{w}$ $(a, \varepsilon, \varepsilon)(c, d, \varepsilon)(\varepsilon, \varepsilon, b)(a, a, c)$
w $\underbrace{(a, \varepsilon)(c, \varepsilon)}(\varepsilon, b)(a, c)$
u $(d, \varepsilon)(\varepsilon, b)(a, c)$
w' $(\varepsilon, b)\overbrace{(a, \varepsilon)(d, \varepsilon)}(a, c)$
u' $(\varepsilon, b)(d, \varepsilon)(a, c)$
$\mathbf{w}'$ $(\varepsilon, \varepsilon, b)(a, \varepsilon, \varepsilon)(a, d, \varepsilon)(a, a, c)$

Illustration of $[v \circ u_v] \subseteq v \circ [u_v]$, for $v = (a, \varepsilon)(c, d)(a, a)$.

The discussion above and the diagram convey the intuition of why the result holds, yet we do not wish to introduce a formal framework for handling commutativity. So we use Algorithm 1 to provide a formal proof of $[V \circ u_v] \subseteq V \circ [u_v]$ in the case of words, based directly on the definition of equivalence via morphisms Φ_1 and Φ_2. In this algorithm, the two variables k_1 and k_2 have no real use, but we use them to specify invariants.

The invariants preserved by the loop are the followings:

$$
\begin{aligned}
&k_v = |\pi_1(w'[1..k])| \\
&k_1 = |\pi_1(v[1..k_v])| \\
&k_2 = |\pi_2(w'[1..k_v])| \\
&v[1..k_v] = \pi_{1,2}(\mathbf{w}'[1..k]) \\
&u'[1..k'] = \pi_{2,3}(\mathbf{w}'[1..k]) \\
&(\Phi_1(u'))\,[1..k'] = (\Phi_1(u))\,[1..k_1] \\
&(\Phi_2(u'))\,[1..k'] = (\Phi_2(u))\,[1..k_2] \\
&k' = |\pi_{1,3}\left((v^{-1} \bowtie w')\,[1..k]\right)|.
\end{aligned}
$$

Consequently, after the last iteration,

$$
\begin{aligned}
&k_v = |v| \\
&k_1 = |\Phi_1(u)| \\
&k_2 = |\Phi_2(u)| \\
&k = |\mathbf{w}| = |w'| \\
&k' = |\pi_{1,3}\left((v^{-1} \bowtie w')\right)| = |u|.
\end{aligned}
$$

Algorithm 1: Auxiliary algorithm for the proof of Lemma 5.6

```
**Input**: $v$, $u$, and $w'$
**Output**: $u'$ and $\mathbf{w}'$
Initialization
$k_v = 0$; $k' = 0$; $k_1 = 0$; $k_2 = 0$
//Loop over $w'$
**for** $k = 0$ **to** $|w'|$ **do**
    **switch** $w'[k]$ **do**
        **case** $(\varepsilon, b)$
            $k'$++; $k_2$++
            $u'[k'] = (\varepsilon, b)$
            $\mathbf{w}'[k] = (\varepsilon, \varepsilon, b)$
        **case** $(a, c)$
        // implies $v[k_v] \in \Sigma \times \Sigma$ after $k_v$'s increment
            $k'$++; $k_v$++; $k_1$++; $k_2$++
            Let $(b, c) = v[k_v]$ in
            $u'[k'] = (b, c)$
            $\mathbf{w}'[k] = (a, b, c)$
        **case** $(a, \varepsilon)$
            $k_v$++
            **if** $v[k_v] = (a, \varepsilon)$ **then**
                $\mathbf{w}'[k] = (a, \varepsilon, \varepsilon)$
            **else**
                let $(a, b) = v[k_v]$ in
                $k'$++; $k_1$++
                $u'[k'] = (b, \varepsilon)$
                $\mathbf{w}'[k] = (a, b, \varepsilon)$;
            **end**
    **end**
**end**
```

This concludes our proof. □

Thus, by item 1, given an editing script u_s and a view V, $V^{-1} \bowtie u_s \bowtie V$ consists of a single tree t' (or is empty). We can also observe that $\pi_{1,4}(t') = V^{-1} \circ u_s \circ V$. This leads to the following definition, illustrated on Figure 5.4.

Definition 5.5. *Given an editing script u_s and a view V, the* editing script induced *by u_s on the view V is the editing script $V^{-1} \circ u_s \circ V$.*

We do not want updates to affect the visibility of nodes. It seems to us that it would make little sense to translate an insertion on the view by showing a hidden node. This assumption is justified among others by the potential information leaks induced by such behaviour in an access control framework. Similarly, deleting a node should result in proper deletion, and not in hiding that node. It should be noted, however, that those updates are not always considered irrelevant. Keller [Kel85] for instance explicitly allows this kind of updates.

Definition 5.6. *An editing script u_s is* stable *w.r.t. view V if $V^{-1} \bowtie u_s \bowtie V$ is non empty and if no node of that tree $V^{-1} \bowtie u_s \bowtie V$ has label in $\{\varepsilon\} \times \Sigma \times \Sigma \times \Sigma$ or $\Sigma \times \Sigma \times \Sigma \times \{\varepsilon\}$.*

Let us note that the set of stable editing scripts w.r.t. a regular view V is regular as defined by $\pi_{2,3}(V^{-1} \bowtie T_{\Sigma_{\text{edit}}} \bowtie V \cap \text{Correct})$ where Correct is the regular set of tree alignments over the alphabet $\Sigma_{\text{edit},4}$ with no occurrences of $\{\varepsilon\} \times \Sigma \times \Sigma \times \Sigma$ or $\Sigma \times \Sigma \times \Sigma \times \{\varepsilon\}$. A VPA representation for the set of all stable editing scripts can be computed in polynomial (quadratic) time from the VPA representation of V. Furthermore, we note that the set of stable scripts is closed under $\sim$.

Lemma 5.7. *For every editing scripts u_v, every stable editing script u_s, for every view V, the following assumptions are equivalent:*

$$\pi_{1,4}(V^{-1} \bowtie u_s \bowtie V) \sim u_v \tag{5.1}$$

$$\textit{iff} \;\; u_s \in \pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1}) \tag{5.2}$$

$$\textit{iff} \;\; u_s \circ V \in [V \circ [u_v]] \tag{5.3}$$

$$\textit{iff} \;\; u_s \circ V \in V \circ [u_v] \tag{5.4}$$

Proof. (5.3) ⇔ (5.4) by Proposition 5.6 item 2. Let us prove (5.1) ⇔ (5.2): Let ψ be the (bijective) morphism that relabels a node with label $(\alpha, \beta, \gamma, \delta)$ into a node with label $(\beta, \alpha, \delta, \gamma)$. Then, for every tree alignment t over $\Sigma_{\text{edit},4}$, if $t = V^{-1} \bowtie u_s \bowtie V$ and $\pi_{1,4}(V^{-1} \bowtie u_s \bowtie V) \sim u_v$, then $\psi(t) \in V \bowtie [u_v] \bowtie V^{-1}$ and $u_s = \pi_{1,4}(\psi(t))$. Conversely, if $t \in V \bowtie [u_v] \bowtie V^{-1}$ and $u_s = \pi_{1,4}(t)$, then $\psi^{-1}(t) = V^{-1} \bowtie u_s \bowtie V$ and $\pi_{1,4}(\psi^{-1}(t)) \sim u_v$. To conclude, we prove

$$\begin{tikzcd}
{} \arrow[r, dotted, "\text{source update } u_s"] \arrow[d, "V"'] & {} \arrow[d, "V"] \\
{} \arrow[r, "\text{induced view update } u_v"'] & {}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
{} \arrow[r, dotted, "\text{Propagations of } u_v"] \arrow[d, "V"'] & {} \arrow[d, "V"] \\
{} \arrow[r, "\text{view update } u_v"'] & {}
\end{tikzcd}$$

$$u_v = V^{-1} \circ u_s \circ V \qquad\qquad \begin{array}{c} \forall u_s \in \mathrm{Prop}\,(V, u_v), \\ u_v = V^{-1} \circ u_s \circ V \end{array}$$

Figure 5.4.: The view update framework: induced script and propagations.

(5.2) $\implies$ (5.4) and (5.4) $\implies$ (5.1). If $u_s \in \pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1})$, Then, $u_s \circ V \in V \circ [u_v] \circ V^{-1} \circ V$. Hence, $u_s \circ V \in V \circ [u_v]$, so (5.2) $\implies$ (5.4). Conversely, if $u_s \circ V \in V \circ [u_v]$, then $V^{-1} \circ u_s \circ V \in V^{-1} \circ V \circ [u_v]$. Hence $V^{-1} \circ u_s \circ V \in [u_v]$, so (5.4) $\implies$ (5.1). □

5.1.3. Propagation of a View Update

Given an update on the view, we want to define which propagations on the source we allow. Roughly speaking, we will require a *propagation* of a view update to be side-effect free, i.e., induce the update defined by the user on the view, and to preserve visibility of nodes. Of course we also want our propagations to be schema compliant, i.e., we want the resulting source document to follow the source document schema. But this will be enforced by the definition of the view: we assume the domain of the view to be exactly the source document schema. Figure 5.4 illustrates the following definition for propagations.

Definition 5.7. *An editing script u_s is a* propagation *of editing script u_v w.r.t. view V iff u_v is equivalent to the editing script induced by u_s on the view V, and u_s is stable w.r.t. V. We denote by Prop$\,(V, u_v)$ the set of all propagations of u_v w.r.t. view V.*

We extend the definition to sets as usual: given a set of editing scripts L, $\mathrm{Prop}\,(V, L) = \bigcup_{u_v \in L} \mathrm{Prop}\,(V, u_v)$. Given a document $t \in T_\Sigma$, a view V and an editing script u_v, we may wish to compute an automaton representing all the propagations of u_v from t, i.e., $\{t' \in \mathrm{Prop}\,(V, u_v) \mid \pi_1(t') = t\}$. This can be achieved in polynomial time according to the following proposition.

Proposition 5.8. *Given a view V and an editing script u_v, we can compute in polynomial time an automaton for the set Prop$\,(V, u_v)$.*

Proof. We first compute an automaton for the set $[u_v]$, and then pick up the stable editing scripts from $\pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1})$. Equivalently, if we denote by L the set of all trees in $(V^{-1} \bowtie T_{\Sigma_{\text{edit}}} \bowtie V) \cap \text{Correct}$ such that $\pi_{1,4}(t) \sim u_v$, then $\text{Prop}\,(V, u_v) = \pi_{2,3}(L)$. A VPA for $[u_v]$ already may require size quadratic in u_v, but a detailed analysis of the transitions appearing in the VPA for $\text{Prop}\,(V, u_v)$ shows that the construction is cubic. □

Similarly, one can build in cubic time an automaton that accepts $\text{Prop}\,(V, U_v) = \bigcup_{u_v \in U_v} \text{Prop}\,(V, u_v)$, for every set U_v of editing scripts. Consequently, one can build in cubic time an automaton representing the (source) editing scripts that do not affect a view: those are obtained as the propagations of all "identity" editing scripts.

Proposition 5.9. *Given a view V, we can compute in polynomial time an automaton accepting all (source) editing scripts that keep the view unchanged.*

Proof. These scripts are $\text{Prop}\,(V, T_{\Sigma_{\text{Id}}})$, where $\Sigma_{\text{Id}} = \{(x, x) \mid x \in \Sigma\}$. □

Proposition 5.10. *Given a view V and a regular set of editing scripts L, we can decide in polynomial time if all editing scripts in L keep the view unchanged.*

Proof. Of course we could test the inclusion of L in the set of editing scripts that keep the view unchanged, but this would be unefficient, so we use the direct approach. The editing scripts in L keep the view unchanged if and only if no tree in $V^{-1} \circ L \circ V$ contains a node with label outside of $\{(x, x) \mid x \in \Sigma\}$, which can clearly be tested in polynomial time. □

Remark 5.4. *In particular, this yields a polynomial algorithm to solve the query- (or view-) update independence problem for regular update functions.*

One may wish to select a unique propagation from t. We outline two approaches that help select a propagation. As a first approach we propose to optimize the propagation with respect to a cost function such as edit distance. And as a second approach we propose to use typing mechanisms in order to eliminate undesirable propagations. Finally, none of those two approaches guarantee that a single propagation will be selected, but a unique propagation can be defined by indicating preferences.

Computing the Optimal Propagations Let L denote the language $\{t' \in \text{Prop}\,(V, u_v) \mid \pi_1(t') = t\}$. We assume the cost of a script t' in L to be the number of nodes from t whose label does not belong to $\{(a, a) \mid a \in \Sigma\}$: $\text{cost}(t') = |\{n \in N_t \mid \exists \alpha \neq \beta . lab_t(n) = (\alpha, \beta)\}|$. We must find the editing scripts of minimal cost in L, where L is given by a VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$. Let t' an editing script of minimal cost in L. Proposition 3.12 implies that the depth of t' is bounded by the number of pairs of states in the automaton, by minimality of t' and because $\pi_1(t') = t$. The same bound holds for the number of children of every node in t'. This observation suggests an algorithm that computes for each pair of states $q, q' \in Q$ a hedge of minimal cost accepted by $\mathcal{A}_{q,q'}$. Actually, computing a tree of minimal cost would be too expensive due to the fact that the smallest tree accepted by an automaton or even a DTD may be of exponential size (consider the DTD D_n defined by the rules $r \to a_n$ and $a_i \to a_{i-1}a_{i-1}$ for all $i \geqslant 1$, and $a_0 \to a$). However, we can compute in polynomial time this minimal cost as well as an automaton representation of the trees of minimal cost. We will not detail further the computation of optimal propagations in this dissertation, but some exploratory work on the topic was published in [SBG10] in a restricted setting.

Typing Typing the nodes may also help to reduce the number of propagations. We can use types and require that propagations do not change the types of nodes that are preserved by the view update u_v. Formally, a document *typing* is a function Θ which maps a tree t to a function $\Theta_t : N_t \to \Gamma$, where Γ is a set of types. A propagation S' of a view update S *preserves* Θ-*typing* iff for every $n \in N_{S'}$ with label in Σ^2, we have $\Theta_{\pi_1(S')}(n) = \Theta_{\pi_2(S')}(n)$. The typing could be based on rich schema formalisms, like EDTD. Another possibility would be to base the typing on the states of the automaton used to verify that the sequence of children is valid w.r.t. the DTD. It would require the automata to be deterministic (or at least unambiguous), but XML DTD are usually required to be deterministic, anyway. We addressed these problems regarding the propagation of a view update in [SBG10]. This paper specialized the view update problem for annotated DTDs with simple annotations. Thus, the access specifications considered did not use any complex filters. As a consequence, the visibility of nodes was defined locally, which allowed a much simpler representation of the propagations through propagation graphs, and a shortest path algorithm in those graphs yielded the optimal propagations. On the other hand, annotated DTDs are more restricted in terms of expressive power than our views that have full *MSO* expressivity and can relabel nodes, so that the above presentation is slightly more complex, but also more general.

5.2. Update Functions

As seen before, an editing script u defines an update on its input tree $\pi_1(u)$. The notion of update function generalizes this, as an update function defines how a set of input trees should be updated.

Definition 5.8. *A set of editing scripts f is an* update function *iff for all $u, u' \in f$, $\pi_1(u) = \pi_1(u')$ implies $u \sim u'$. The set of trees $\{\pi_1(u) \mid u \in f\} \subseteq T_\Sigma$ is called the* domain of *f.*

Note that if L is an update function, then the induced relation $\{(\pi_1(u), \pi_2(u)) \mid u \in L\}$ is functional, but the converse is false. In this paper, we are only interested in update functions that are regular sets. These have a finite representation by means of tree automata over Σ_{edit}. The following proposition adapts to our setting a classical property, namely that testing equivalence of functional transductions can be reduced to testing functionality of the union of those transductions. It states that two update functions are equivalent if and only if they have the same domain and their union is an update function.

Proposition 5.11. *Two update functions f_1 and f_2 are equivalent if and only if $\pi_1(f_1) = \pi_1(f_2)$ and $f_1 \cup f_2$ is an update function.*

Proof. For the *only if* part suppose $f_1 \sim f_2$. Then $\pi_1(f_1) = \pi_1(f_2)$. Let $u_1 \in f_1, u_2 \in f_2$ such that $\pi_1(u_1) = \pi_1(u_2)$. By hypothesis, there exists $u_1' \in f_1$ such that $u_1' \sim u_2$. Since $\pi_1(u_1) = \pi_1(u_2) = \pi_1(u_1')$, $u_1 \sim u_1'$ (f_1 is an update function). Hence, $u_1 \sim u_2$. Therefore, $f_1 \cup f_2$ is an update function.

For the *if* part, suppose $\pi_1(f_1) = \pi_1(f_2)$ and $f_1 \cup f_2$ is an update function. Then for every $u_1 \in f_1$, there exists $u_2 \in f_2$ such that $\pi_1(u_1) = \pi_1(u_2)$. By hypothesis, $u_1 \sim u_2$. Therefore $f_1 \sim f_2$. □

Proposition 5.12. *The composition $f_1 \circ f_2$ of two update functions f_1 and f_2 is an update function.*

Proof. Fix $s, s' \in f_1 \bowtie f_2$. There are $u_1, u_1' \in f_1, u_2, u_2' \in f_2$ such that $s \in u_1 \bowtie u_2$ and $s' \in u_1' \bowtie u_2'$. Suppose $\pi_1(s) = \pi_1(s')$. Then $\pi_1(u_1) = \pi_1(u_1')$, hence $u_1 \sim u_1'$ (f_1 is an update function). Consequently, $\pi_1(u_2) = \pi_2(u_1) = \pi_2(u_1') = \pi_1(u_2')$, hence $u_2 \sim u_2'$. Thus, $u_1 \sim u_1'$ and $u_2 \sim u_2'$. We conclude the proof using propositions 5.4 and 5.5: $\pi_1(s) = \pi_1(s')$ implies $s \sim s'$, so $f_1 \circ f_2$ is an update function. □

Without intersecting $\pi_{1,3}(L_1 \bowtie L_2)$ with L_{corr} in the definition of composition, this property would not hold: given $f_1 = (a, \varepsilon)$ and $f_2 = (\varepsilon, b)$, $\pi_{1,3}(L_1 \bowtie L_2)$ comprises (a, b), $(a, \varepsilon)(\varepsilon, b)$ and $(\varepsilon, b)(a, \varepsilon)$, hence is not an update function.

5.2.1. Functionality and Disambiguation

Proposition 5.13. *Given a regular set L of editing scripts, it is decidable whether L is an update function in time polynomial in the size of the automaton defining L.*

Proof. According to Proposition 5.1, the language $L^{-1} \bowtie L$ is regular, so its linearization is context-free. Furthermore, the morphisms we have defined on tree alignments (projections, inversions and Φ_1, Φ_2) can be viewed as word homomorphisms on the linearizations. We define morphisms f_1, f_1', f_2, f_2' on trees over $\Sigma_{\text{edit},3}$ as: $f_1 : t \mapsto \Phi_1\left((\pi_{1,2}(t))^{-1}\right)$, $f'_1 : t \mapsto \Phi_1\left(\pi_{2,3}(t)\right)$, and similarly for f_2, f'_2 with Φ_2 instead of Φ_1. From the equality $\{(s,s') \in L^2 \mid \pi_1(s) = \pi_1(s')\}$=$\{((\pi_{1,2}(t))^{-1}, \pi_{2,3}(t)) \mid t \in L^{-1} \bowtie L\}$ we get $f_1(t) = f'_1(t)$ for every $t \in L^{-1} \bowtie L$ iff $\Phi_1(t_0) = \Phi_1(t'_0)$ for every t_0 and $t'_0 \in L$ such that $\pi_1(t_0) = \pi_1(t'_0)$. We obtain the same condition for f_2, f'_2 and Φ_2 and this implies $f_1(t) = f'_1(t)$ and $f_2(t) = f'_2(t)$ for every $t \in L^{-1} \bowtie L$ iff $t_0 \sim t'_0$ for every $t_0, t'_0 \in L$ such that $\pi_1(t_0) = \pi_1(t'_0)$. Therefore it suffices to use Plandowski's result that equivalence of morphisms on context-free languages is decidable in polynomial time [Pla94] in order to verify that $t \mapsto \Phi_1\left((\pi_{1,2}(t))^{-1}\right)$ and $t \mapsto \Phi_1\left(\pi_{2,3}(t)\right)$ are equivalent morphisms on $L^{-1} \bowtie L$, and similarly for Φ_2.□

Proposition 5.13 actually states a decidability result for the functionality of a particular kind of transduction. The "squaring" technique used in the proof is fairly standard, and already appears in [SLLN09, FRR+10] to decide functionality for some models of visibly pushdown transducers. The technique can be generalized to test k-valuedness of a transduction for arbitrary k, functionality corresponding to 1-valuedness. Functionality is tested by computing an automaton accepting trees representing in parallel two instances of the transduction over a same input. The alphabet of this "square" automaton is therefore Σ^3, or Σ^2 since we can project out the input component. Once this square automaton has been computed, one must check if it accepts a word representing two distinct transductions. Several techniques exist: one can use results on morphisms such as Plandowski's, or one can use results on reversal-bounded multicounter machines such as Ibarra's. Actually both Plandowski's [Pla94] and Gurari and Ibarra's [GI81] rely on a pumping argument. The first decidability result for testing functionality of two word transducers directly used a pumping lemma to bound the size of the input word on which the transducer may produce two different outputs by a quadratic polynomial in the size of the transducer [Sch75]. Recently, Filiot et al. [FRR+10] applied and improved similar techniques to decide k-functionality of visibly pushdown transducers in NP.

When the user formulates an update function on its view, the resulting set of all propagations on the source may be ambiguous. Disambiguating a set of updates, i.e., making it functional, is a key point as eventually only one specific update will be applied to the document. The following theorem shows how this inherent ambiguity of update translation can be resolved by arbitrary choices while preserving the regularity of the update.

Theorem 5.14. *Given a regular set of editing scripts L, we can effectively compute a regular update function L' such that $L' \subseteq L$ and the domains of L' and L are equal $(\pi_1(L) = \pi_1(L'))$.*

Proof. In this proof we represent the regular languages of editing scripts L with an NTAs $A = (\Sigma_{\text{edit}}, Q, Q_f, \Delta)$ whose language is the *fcns* binary encoding of the tree alignments.

Let $h = (t_1, \ldots, t_n)$ a hedge -eventually empty- of closed trees. We will here abusively confuse h and the *fcns* encoding of $(t_1, \ldots, t_n)$. We denote by (ε, h) the encoding of $\pi_2^{-1}(h) \cap \pi_1^{-1}(\varepsilon)$. To get rid of insertions, we extend the NTA model to allow rules of the form $(f, h_1, \alpha, h_2)(q_1, q_2) \to q$ where $q_1, q_2, q \in Q$, $f \in \Sigma$, $\alpha \in \Sigma_\varepsilon$ and h_1, h_2 are (*fcns* encoding of) hedges over Σ; if $\alpha = \varepsilon$, h_1 has to be empty. The semantics of such a rule is to rename the node labelled by f with α, insert h_1 ahead of its descendants and h_2 after the node. Formally, a rule $(f, h_1, \alpha, h_2)(q_1, q_2) \to q$ can be applied to assign state q to some node n of *fcns*(t) if and only if

- state q_1 has been assigned to the representant n' in *fcns*(t) of the first child of n (in t) whose label is not of the form (ε, a) (or, if there is no such child, to the rightmost $\perp$ symbol below the left child of n in *fcns*(t)),
- state q_2 has been assigned to the representant n'' in *fcns*(t) of the closest following sibling of n whose label is not in $\{\varepsilon\} \times \Sigma$ (or, if there is no such sibling, to the rightmost $\perp$ symbol below n in *fcns*(t)),
- the children of n until n' form the hedge (ε, h_1), and
- and the siblings of n until n'' form the hedge (ε, h_2)

We observe that a rule of the form $(f, \alpha)(q_1, q_2) \to q$ can be viewed as a rule $(f, h_1, \alpha, h_2)(q_1, q_2) \to q$ with empty hedges h_1, h_2. A rule $(f, h_1, \alpha, h)(q_1, q_2) \to q$ can be viewed as the composition of the rule $(f, \alpha)(r_1, r_2) \to q$ with $(\varepsilon, h_1)(q_1) \to^* r_1$, $(\varepsilon, h_2)(q_2) \to^* r_2$. We explain next how we associate with the tree alignment automaton A an extended automaton B s.t. has

- property $P1$ if $L(B) \subseteq L(A)$
- property $P2$ if $\pi_1(L(B)) = \pi_1(L(A))$

Elimination of insertions: We suppose that every state $q \in Q$ is productive, i.e., there is at least one tree accepted by the NTA $(Q, \{q\}, \Delta)$.

- for every rule $(\varepsilon, a)(q_1, q_2) \to q$, according to the definition of editing scripts, q_1 accepts only trees whose first component is entirely labelled by ε . For any such state q, we choose arbitrarily a tree accepted by q and note t_q its image by π_2.
- we add a rule $(a, h_1, \alpha, h_2)(q_p, r_n) \to q$ with

  $h_1 = \beta_1(t_{l_1}, \beta_2(t_{l_2}, \ldots, \beta_p(t_{l_p}) \ldots))$,

$h_2 = \alpha_1(t_{s_1}, \alpha_2(t_{s_2}, \ldots, \alpha_n(t_{s_n})\ldots))$

for every $n, p \in \{0, \ldots, |Q|\}$, and for every sequence of rules of the form:

$(a, \alpha)(q_0, r_0) \to q$, ($a$ in Σ)

$(\varepsilon, \alpha_1)(s_1, r_1) \to r_0$

$(\varepsilon, \alpha_2)(s_2, r_2) \to r_1$

$\vdots$

$(\varepsilon, \alpha_n)(s_n, r_n) \to r_{n-1}$,

$(\varepsilon, \beta_1)(l_1, q_1) \to q_0$

$(\varepsilon, \beta_2)(l_2, q_2) \to q_1$

$\vdots$

$(\varepsilon, \beta_p)(l_p, q_p) \to q_{p-1}$, with $p = 0$ if $\alpha = \varepsilon$,

- we eliminate rules labelled with (ε, α) and (a, α);

Disambiguation: let us notice that, as we have eliminated transitions labelled with (ε, α), we get from an extended alignment automaton $\mathcal{A}$ an "usual" tree automaton $\mathcal{A}_1$ over Σ by "forgetting" the second component i.e. associating with a rule $(f, h_1, \alpha, h_2)(q_1, q_2) \to q$ the rule $f(q_1, q_2) \to q$. We will call A_1 this automaton. Actually, the following construction relies on disambiguation of this automaton A_1.

First, we transform A_1 while keeping Properties P_1 and P_2 to make sure that for each letter a and triple q_1, q_2, q there is at most one rule of the form $(a, h_1, \alpha, h_2)(q_1, q_2) \to q$ in A. This can easily be achieved by removing some transitions. We call δ the resulting transition function. Secondly, we choose a total order $<$ on $Q \times Q$, e.g. lexicographic order induced by a total order on Q. And finally, we define $B_1 = (Q', \Delta', Q'_f)$ defined by:

- $Q' = \{(q, S, F) \mid S \subseteq Q, F \subseteq Q, q \in S, S \cap F = \varnothing\}$
- Δ' contains the rule $(f, \alpha, h)((q_1, S_1, F_1), (q_2, S_2, F_2)) \to (q, S, F)$ Iff
  - $F \cup S = \delta(f, F_1 \cup S_1, F_2 \cup S_2)$
  - $\delta(f, F_1, S_2 \cup F_2) \cup \delta(f, F_1 \cup S_1, F_2) \subseteq F$: a fail leads only to fails.
  - $\forall s_1 \in S_1, s_2 \in S_2, \delta(f, s_1, s_2) \cap S \neq \varnothing$: a success leads at least to one success.
  - $\forall (q'_1, q'_2) \in S1 \times S2, q \in \delta(f, q'_1, q'_2) \implies (q'_1, q'_2) > (q_1, q_2)$: this guarantees minimality (unicity) of the run.
  - $(f, \alpha, h)(q_1, q_2) \to q$ is a rule of A : this guarantees the transformation satisfies P_1

$$\begin{tikzcd}
{} \arrow[r, dotted, "\text{translation}"] \arrow[d, "V"'] & {} \arrow[d, "V"] \\
{} \arrow[r, "\text{update}"'] & {}
\end{tikzcd}$$

Figure 5.5.: The view update problem.

- $Q'_f = \{(q, S, F) \mid S \subseteq Q_f, F \cap Q_f = \emptyset, q = min(S)\}$

We check easily that $t \rightarrow^*_{B_1} (q, S, F)$ only when $t \rightarrow^*_{A_1} S \cup F$. Then, it is easy to check uniqueness of accepting run because there is no choice for the node labelling the root of an accepting run, and therefore by induction no choice for the states labelling the other nodes. Properties P_1 and P_2 are obviously preserved by the construction. □

Theorem 5.14 gives a theoretical solution for disambiguating a regular set of editing scripts. The construction is not polynomial but could be applied on the fly to increase efficiency: rather than first constructing an unambiguous set of editing scripts and then apply it on the document, we can disambiguate on the fly by a two-pass run on the document.

Theorem 5.14 states that one can uniformize relations defined by editing scripts. Uniformization is a well-studied property for word transducers (see [CG99] for a survey). Filiot and Servais prove a related result in [FS11]: they prove that for every visibly pushdown transducer T, one can compute a deterministic visibly pushdown transducer T' with regular lookahead such that the domains of T' and T are equal and the transduction defined by T' is a subset of the transduction defined by T'. Consequently, the transductions defined are equal when T is functional. The authors order the states as we do in the proof above, then extend this ordering to runs, and use the lookahead to obtain a "minimal" run.

5.2.2. Update Translation

We extend the notion of propagation to sets of editing scripts: given a set of editing scripts L, the propagations of L are defined by $\text{Prop}(V, L) = \{\text{Prop}(V, u_v) \mid u_v \in L\}$. Now, for an update function f_v on the view V, we want to characterize which sets of editing scripts can be considered as correctly and completely propagating f_v on the source:

Definition 5.9. *Given an update function f_v, and a set of editing scripts L, we say that L is a* translation *of update f_v w.r.t. view V if L consists of stable editing scripts w.r.t. view V and $L \circ V \sim V \circ f_v$.*

Thus, a set of editing scripts is a translation if the diagram of Figure 5.5 commutes. As we could expect, propagations and translations (as well as stability) are preserved under equivalence:

Remark 5.5. *Observe that, by Proposition 5.5, an editing script equivalent to a propagation is a propagation, and a set of editing scripts equivalent to a translation is a translation.*

There is an alternative characterization of translations:

Proposition 5.15. *L is a translation of update function f_v iff $L \subseteq Prop\,(V, f_V)$ and $\pi_1(V \circ f_v) \subseteq \pi_1(L \circ V)$.*

Proof. Let L a translation of update function f_v and u_s in L; as $L \circ V \sim V \circ f_v$, $u_s \circ V \sim v \circ u_v$ for some u_v in f_v and v in V; then $V^{-1} \circ u_s \circ V \sim u_v$ according to Lemmma 5.7: so $L \subseteq \text{Prop}\,(V, f_V)$; furthermore as $L \circ V \sim V \circ f_v$, $\pi_1(L \circ V) = \pi_1(V \circ f_v)$.

Conversely, let L s.t. $L \subseteq \text{Prop}\,(V, f_V)$ and $\pi_1(L \circ V) = \pi_1(V \circ f_v)$. As $L \subseteq \text{Prop}\,(V, f_V)$, L consists of stable editing scripts w.r.t. V and $V^{-1} \circ L \circ V \subseteq [f_v]$. So $L \circ V \subseteq V \circ [f_v]$ by Lemma 5.7, using (5.1) $\implies$ (5.4). Then, by Lemma 5.6 item 2, $L \circ V \subseteq [V \circ f_v]$ as L consists of stable editing scripts. As $\pi_1(V \circ f_v) \subseteq \pi_1(L \circ V)$ and $V \circ f_v$ is functional, $L \circ V \sim V \circ f_v$. □

The Update Translation Problems

Problem 1 (Checking a translation). *Given a regular view V, a regular view update function f_v, and a regular set of source editing scripts L_s, answer whether L_s is a translation of f_v.*

Problem 2 (Finding a translation). *Given a regular view V and a regular view update function f_v, find a regular set of source editing scripts L_s s.t. L_s is a translation of f_v.*

5.2.3. Solution in the Unconstrained Case

From now on, we suppose w.l.o.g. that $\pi_1(f_v) \subseteq \pi_2(V)$. We further assume that $\pi_2(f_v) \subseteq \pi_2(V)$ otherwise there would be no translation for f_v. Every update function satisfying those requirement is translatable. These assumptions are reasonable insofar as we can suppose the user to be provided a view schema. Besides, one can verify those assumptions in time polynomial in terms of f_v and exponential in terms of V (this problem of regular tree languages inclusion is of course EXPTIME-complete). Proposition 5.16 answers Problem 1 positively.

Proposition 5.16. *Given a regular view V, a regular update function f_v, and a regular set of source editing scripts L, testing whether L is a translation of f_v is decidable.*

Proof. First, we test whether L consists of stable updates. Next, we must check that $L \circ V \sim V \circ f_v$. We claim that $L \circ V \sim V \circ f_v$ iff $\pi_1(L \circ V) = \pi_1(V \circ f_v)$ and $L \circ V \cup V \circ f_v$ is an update function. Once we have tested the equality of the domains, namely $\pi_1(L \circ V) = \pi_1(V \circ f_v)$, we can use Proposition 5.13 and check that $L \circ V \cup V \circ f_v$ is an update function. Let us prove the claim: $V \circ f_v$ is an update function, by Proposition 5.12. Now, either $L \circ V$ is not an update function and then it is not equivalent to $V \circ f_v$, but $L \circ V \cup V \circ f_v$ is not an update function either. Or $L \circ V$ is an update function and the claim follows by Proposition 5.11. This concludes our proof. Furthermore, the algorithm is polynomial once we have checked equality of the domains. □

The following proposition answers Problem 2 positively.

Proposition 5.17. *Given a regular view V and a regular update function f_v, we can compute a translation L of f_v in polynomial time.*

Proof. By Propositions 5.1 and 3.18, we can compute in polynomial time an automaton for the set L of all editing scripts from $\pi_{1,4}(V \bowtie f_v \bowtie V^{-1})$ that are stable (w.r.t. V). We must show that L is a translation of f_v. By Lemma 5.7, using (5.1) $\implies$ (5.2), L consists of propagations of f_v. The above assumptions ensure that $\pi_1(L \circ V) \subseteq \pi_1(V \circ f_v)$. □

Finally, using Theorem 5.14, we get

Corollary 5.18. *We can compute a functional translation L of f_v.*

5.3. Translating Update Functions Under Constraints

Let us resume with the illustrative example from section 5.1. We assume that once a project has acquired the `'stable'` status, it cannot be modified anymore, so that no `'stable'` project should revert to the `'dev'` status. This in turn implies that authority A_1 cannot delete certificates c_1 under a document that has also been certified by the second authority. Such constraints can clearly be expressed via a regular set of editing scripts. However, if we do only forbid the above deletions, A_1 may face a strange behavior since it does not know about certificates c_2. Thus, A_1 will observe that it is sometimes allowed to delete its certificate c_1 under some `documents` nodes, and sometimes not. The uniform updates are those that avoid this kind of unpredictable behaviour. Here, the uniform updates forbid deleting any c_1 certificate altogether. Computing the set of uniform updates enables the database administrator to provide the user with the set of updates she is allowed to execute, which is the motivation for Problem 5.

5.3.1. The General Case

Our views impose only static constraints on the state of the database. We wish to study constraints on the updates in the spirit of the *transition laws* of [FUV83]. While in [FUV83] the transition laws are treated as static constraints, using an extended database, our approach focuses on studying transitions, and the constraints we define on the updates cannot be expressed by static constraints within our framework.

In this section, we suppose a given regular set of editing scripts $\mathcal{U}_s$ representing the authorized source updates. Furthermore, we are going to consider only translations valid w.r.t. $\mathcal{U}_s$ (as formalized by Definition 5.10). Such restrictions can be most useful in the case of a database with multiple user profiles. One may require for instance that the updates of user 1 should not affect the view of user 2, or more permissively, the updates of user 1 should affect user 2's view only on nodes that are also visible in user 1's own view. A regular set $\mathcal{U}_s$ of authorized source updates can express that kind of restrictions on side effects. This approach is more flexible than the constant complement approach of [BS81] in the sense that we do not require the constant part to be a complement. Thus, the user can specify precisely the constraints he deems relevant, without the obligation to enforce a unique propagation. Such restrictions can also be used to protect the integrity of sensible data or to indicate some preference among possible propagations, as demonstrated in Theorem 5.32, in order to get a unique propagation. More generally, the possibility to define a set of authorized source updates allows the database administrator to specify which updates he thinks are reasonable.

Definition 5.10. *A set of editing scripts L is a* valid translation *of an update function f_v w.r.t. view V and set $\mathcal{U}_s$ if L is a translation of f_v w.r.t. V and there exists a set of editing scripts $L' \subseteq \mathcal{U}_s$ s.t. $L' \sim L$.*

A view editing script is called uniform *if it admits a valid translation. We denote by $Unif\,(V,\mathcal{U}_s)$ the set of uniform (view) editing scripts.*

An update function f_v is called uniformly translatable *w.r.t. view V and $\mathcal{U}_s$ if it has a valid translation.*

Let us note that even when f_v and V are regular, we impose in the definition neither regularity of L nor regularity of L'.

Example 5.3. *Let V denote the identity and $\mathcal{U}_s$ denote the set of editing scripts of depth one with yield in $(a,\varepsilon)^*(\varepsilon,b)^*$. The update function f_1 that consists of editing scripts of depth one with yield in $(a,b)^*$ has a regular translation but it does not admit any* valid *translation L. The update function f_2 that consists of editing scripts of depth one with yield in $((a,\varepsilon)(\varepsilon,b))^*$ is uniformly translatable. It is indeed its own valid translation, since the set $L' \subseteq \mathcal{U}_s$ that contains all editing scripts of depth one with yield in $\{(a,\varepsilon)^n(\varepsilon,b)^n \mid n \in \mathbb{N}\}$, though not regular, is equivalent to f_2.*

Proposition 5.19. *An update function f_v is uniformly translatable w.r.t. view V and set $\mathcal{U}_s$ iff there exists some set of stable editing scripts $L \subseteq \mathcal{U}_s$ such that $L \circ V \sim V \circ f_v$.*

Proof. The result is immediate by Remark 5.5. □

However, let us note that the preceding property is no longer valid when we require regularity; f_v can have a regular valid translation but no *regular* valid translation included in $\mathcal{U}_s$, as illustrated by f_2 in Example 5.3. In view of the previous definition and result, the following adapts Problem 1 to the constrained setting.

Problem 3. *Given a regular view V, a regular set of authorized source editing scripts $\mathcal{U}_s$, a regular update function f_v, and a regular set of source editing scripts L, answer if L is a valid translation of f_v.*

While every update function admits a translation in the unconstrained setting, this is no longer the case in presence of constraints. The presence of source constraints raises two additional problems.

Problem 4. *Given a regular view V, a regular set of authorized source editing scripts $\mathcal{U}_s$, and a regular update function f_v, answer if f_v is uniformly translatable.*

Problem 5. *Given a regular view V, and a regular set of authorized source editing scripts $\mathcal{U}_s$, compute an automaton whose language is the set Unif $(V,\mathcal{U}_s)$.*

Negative Results in the General Setting

Proposition 5.20. *Testing uniform translatability is undecidable, even when f_v is regular.*

Proof. Let V be the identity over trees of depth one. Formally, view V is the set of trees of depth one with root (r, r) and yield $(\bigcup_{a \in \Sigma}(a, a))^*$. Suppose f_v uses no relabelings, only deletions and insertions f_v consists only of trees of depth one with root (r, r) and yield in $(\Sigma \times \{\varepsilon\} \cup \{\varepsilon\} \times \Sigma)^*$. Then the problem is equivalent to the problem of testing the inclusion of a functional word transducer (f_v) into an arbitrary word transducer ($\mathcal{U}_s$), which is undecidable [Ber79]. This proves also that testing uniform translatability remains undecidable when we require translation to be regular. The question remains open when we also require regularity of $L' \subseteq \mathcal{U}_s$ such that $L \sim L'$. □

Note that with the same proof, for $L = f_v$, we get the undecidability of Problem 3. However, if the input is some $L \subseteq \mathcal{U}_s$, it becomes decidable in polynomial time once the domains are verified equal, using Proposition 5.16.

Proposition 5.21. *Given a regular view V, Unif $(V,\mathcal{U}_s)$ is not regular, and its emptiness is undecidable.*

Consequently, computing the set of uniform editing scripts seems unfeasible in general, and therefore, we look for restrictions that allow to tackle these problems.

Single Updates The simplest restriction will be to study translatability of single editing script instead of more general update functions. For that limited setting, the previous problems become decidable.

Proposition 5.22. *Testing uniform translatability of a view editing script u_v is decidable. Furthermore, we can compute in polynomial time a regular set L of editing scripts such that $[L]$ is the set of (valid) propagations of u_v.*

Proof. The set of valid propagations is equivalent to $\mathcal{U}_s \cap \pi_{1,4}(V \bowtie [u_v] \bowtie V^{-1})$ by Lemma 5.7, using (5.1)⇔(5.2). Those results can also be considered a consequence of Propositions 5.28 and 5.29. □

The previous results might however be misleading: one could suppose the difficulty to stem from $\mathcal{U}_s$'s not being closed under equivalence. The following undecidability result that holds for $\mathcal{U}_s$ over alphabet $\Sigma_\varepsilon \times \Sigma$ shows it does not. Intuitively, even when $\mathcal{U}_s$ has no deletions, V may have deletions, so that $\mathcal{U}_s \circ V$ needs not be regular.

One could have supposed that solving Problem 5 dynamically rather than statically would be easier: one does not need to compute all the uniform updates, but only those possible from the current state of the (non-materialized) view document. The following proposition puts paid to any such hope. We cannot tackle Problem 5 by fixing the initial document t and asking for the set of all uniform view editing scripts u such that $\pi_1(u) = t$. Fix a tree t over Σ, the tree that consists of a single node r for instance. Even when we require $\mathcal{U}_s$ to consist only in editing scripts without deletions, i.e., trees over $\Sigma_\varepsilon \times \Sigma$, we get the following negative result

Proposition 5.23. *The problem (with input V and $\mathcal{U}_s$) of deciding 'universality' of the set $\{t' \in$ Unif $(V,\mathcal{U}_s) \mid \pi_1(t') = t\}$ (more exactly, testing whether it is equal to the co-domain of the view) is undecidable.*

Proof. Fix a PCP instance $u_1, \ldots, u_n, v_1, \ldots, v_n$ over alphabet Σ. Without loss of generality, we assume that none of $u_1, \ldots, v_n$ are empty words (this variant of PCP is known to be undecidable). We use trees of depth one over alphabet $\Sigma' = \{r, \#1, \ldots, \#n\} \uplus \Sigma$. We define view V as follows: V hides $\#1, \ldots, \#n$ and keeps the other tags unchanged. In order to define $\mathcal{U}_s$, we use two auxiliary languages: we claim that there exists an automaton $\mathcal{A}_u$ accepting some regular set of editing scripts L_u such that $\{(\pi_1(t), \pi_2(t)) \mid t \in$

$L_u \circ V\} = \{(r(\#i_1, \ldots, \#i_k), r(a_1, \ldots, a_m)) \mid i_1, \ldots i_k \leqslant n, m \geqslant 0, a_1 \ldots a_m \in \Sigma^*, a_1 \ldots a_m \neq u_{i_1} u_{i_2} \ldots u_{i_k}\}$. We build L_v symmetrically. $\mathcal{U}_s$ is defined as $L_u \cup L_v$. Consequently, since $\{\pi_1(t) \mid t \in V \wedge \pi_2(t) = r\} = \{r(a_1, \ldots, a_m) \mid m \geqslant 0, a_1, \ldots, a_m \in \{\#1, \ldots, \#n\}\}$, the sets $\{t' \in \mathit{Unif}\,(V, \mathcal{U}_s) \mid \pi_1(t') = r\}$ and $\{(r, r)((\varepsilon, a_1), \ldots, (\varepsilon, a_m)) \mid m \geqslant 0, a_1, \ldots, a_m \in \Sigma^*\}$ are equal if and only if there is no match for the PCP instance.

Let us prove the claim. The construction is similar to the one in [Gri68]. We build $\mathcal{A}_u = (Q, Q_f = \{q_r\}, \Delta)$ on the fcns encoding as follows: Q contains states $q_0, q_r, q_H, q_\perp, q_f$ and q'_f, and, for each word $u_i = a_{i,1} a_{i,2} \ldots a_{i,k_i}$, k_i states $q_{i,1} \ldots q_{i,k_i}$. We define the transitions in Δ as follows: the initial rules are given by $\perp \to q$ for every $q \notin \{q_H, q_0\}$, and the other rules are, for every $i \leqslant n$,

1. for all $a \in \Sigma$, $(\varepsilon, a)(q_\perp, q'_f) \to q'_f$ and $(\varepsilon, a)(q_\perp, q'_f) \to q_H$.
2. for all $j \leqslant n$, all $a \in \Sigma$, $(\#j, \#j)(q_\perp, q_f) \to q_f$ and $(\varepsilon, a)(q_\perp, q_f) \to q_f$.
3. for all $j \leqslant k_i$, for all $a \in \Sigma$ such that $a \neq a_{i,j}$, $(\varepsilon, a)(q_\perp, q_f) \to q_{i,j}$.
4. for every $j < k_i$, $(\varepsilon, a_{i,j})(q_\perp, q_{i,j+1}) \to q_{i,j}$.
5. $(\varepsilon, a_{i,k_i})(q_\perp, q_H) \to q_{i,k_i}$
6. $(\#i, \#i)(q_\perp, q_{i,1}) \to q_0$
7. $q_0 \to q_H$: we use an ϵ-transition here, but it can be eliminated.
8. $(r, r)(q_0, q_\perp) \to q_r$ □

Let $k \geqslant 1$ and $i_1, \ldots, i_k \in \{1, \ldots, n\}$. A word w is different from $u_{i_1} \ldots u_{i_k}$ if and only if one of the following three conditions is satisfied: (i) w is a strict prefix of $u_{i_1} \ldots u_{i_k}$, (ii) $u_{i_1} \ldots u_{i_k}$ is a strict prefix of w or (iii) there exists a common prefix s of w and $u_{i_1} \ldots u_{i_k}$ and two distinct letters x and y such that sx is a prefix of w and sy is a prefix of $u_{i_1} \ldots u_{i_k}$. This proves the correction of automaton $\mathcal{A}_u$: a word w with $\pi_1(w) = i_1 \ldots i_k$ is accepted (i) from some state $q_{i,j}$ if and only if $\pi_2(w)$ is a strict prefix of $u_{i_1} \ldots u_{i_k}$, (ii) from state q'_f if and only if $u_{i_1} \ldots u_{i_k}$ is a strict prefix of $\pi_2(w)$, and (iii) from state q_f if and only if there exists a common prefix s of $\pi_2(w)$ and $u_{i_1} \ldots u_{i_k}$ and two distinct letters x and y such that sx is a prefix of $\pi_2(w)$. This concludes the proof of the claim.

It may be difficult to understand the transition table of such a big automaton, so Figure 5.6 depicts the possible evolution of states from the rightmost child of the tree up to its leftmost child, in the case where $n = 1$ and $u_1 = a_{1,1} a_{1,2} \ldots a_{1,k}$. The "run" can begin from every state except q_H and q_0 (initial rules, represented by $\perp \to q_i$ in the figure), and must end in q_0 (according to rule 8). We have dropped the left $q_\perp$ state of every pair in the picture: a transition $q_i \xrightarrow{a} q_j$ in the picture represents a transition

$a(q_\perp, q_i) \to q_j$ of the tree automaton $\mathcal{A}_u$. When there are several words in the PCP instance (of course, $n > 1$ in general), states q_0, q_H, q_f and q'_f are common to all the u_i's ($i \leqslant n$) but all states $q_{j,i}$ are distinct.

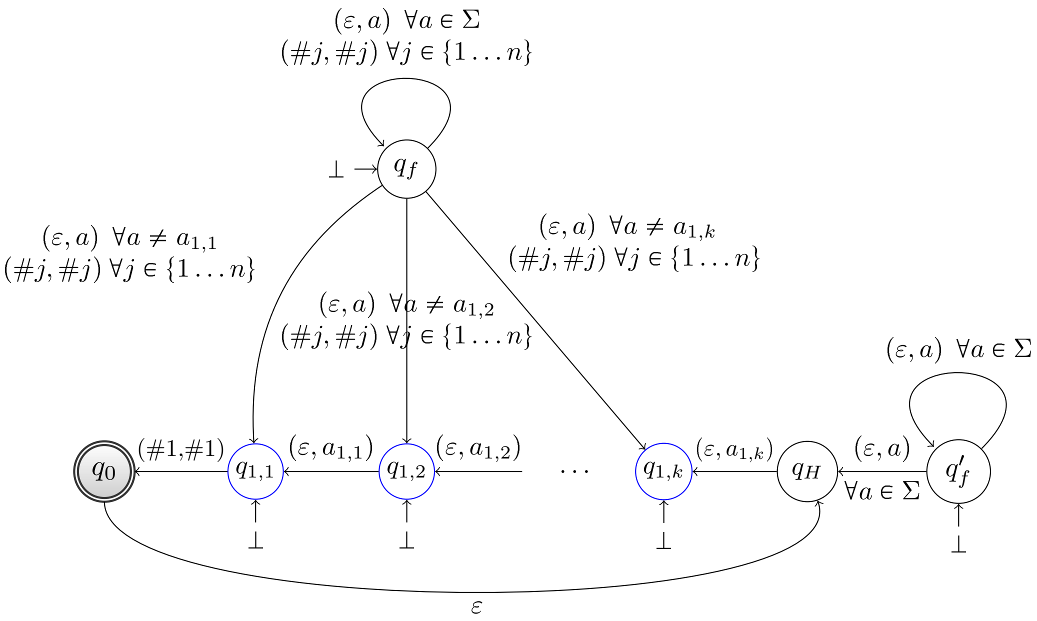

$(\varepsilon, a)\ \forall a \in \Sigma$
$(\#j, \#j)\ \forall j \in \{1 \dots n\}$
$\perp \to$
q_f
$(\varepsilon, a)\ \forall a \neq a_{1,1}$
$(\#j, \#j)\ \forall j \in \{1 \dots n\}$
$(\varepsilon, a)\ \forall a \neq a_{1,2}$
$(\#j, \#j)\ \forall j \in \{1 \dots n\}$
$(\varepsilon, a)\ \forall a \neq a_{1,k}$
$(\#j, \#j)\ \forall j \in \{1 \dots n\}$
$(\varepsilon, a)\ \forall a \in \Sigma$
q_0
$(\#1,\#1)$
$q_{1,1}$
$(\varepsilon, a_{1,1})$
$q_{1,2}$
$(\varepsilon, a_{1,2})$
$\dots$
$q_{1,k}$
$(\varepsilon, a_{1,k})$
q_H
(ε, a)
$\forall a \in \Sigma$
q'_f
$\perp$
$\perp$
$\perp$
$\perp$
ε

Figure 5.6.: "Core" of $\mathcal{A}_u$ for $u_1 = a_{1,1}a_{1,2} \dots a_{1,k}$.

Since our target is a translation of update functions, we look for a less drastic restriction than single updates. What makes translatability decidable for a single (view) editing script u_v is the possibility to compute a regular language for the equivalence class of u_v. The next section defines a class of update functions that guarantees that property, while remaining powerful enough to express most reasonable update functions.

Equivalence of editing scripts deals with commuting consecutive insertions and deletions. For that reason, we must control those commutations to make sure the closure under equivalence of regular editing scripts languages remains regular.

Definition 5.11. *Given a natural $k \geqslant 1$, an editing script t is k-*synchronized *if for every sequence $n_1, n_2, \dots n_{k+1}$ of nodes in N_t such that for all $j \leqslant k$, $(n_j, n_{j+1}) \in next_t$, and for all $j \leqslant k+1$, $lab_t(n_j) \in \{\varepsilon\} \times \Sigma$, there is some node $n' \in N_t$ such that $(n_1, n') \in follow_t$, $(n', n_{k+1}) \in follow_t$, and $lab_t(n') \in \Sigma \times \Sigma$.*

This means that, among the children of the same node, there cannot be more than k inserting nodes without a node tagged with a relabeling between them. A set of editing scripts is *k-synchronized* if it consists of k-synchronized editing scripts. A set of editing scripts is *synchronized* if it is k-synchronized for some k.

Remark 5.6. *This notion is monotone: if a (set of) editing script(s) is k-synchronized, then it is k' synchronized for all $k' > k$.*

General Properties The k-synchronized editing scripts can be viewed as the horizontal counterpart of k-interval bounded view alignments. We can therefore prove similar properties, based on the horizontal pumping arguments.

Proposition 5.24. *Let $\mathcal{A}$ a VPA (resp. NTA) with n states accepting a set of editing scripts (resp. its fcns encoding). If $L(\mathcal{A})$ is synchronized, then it is $n-1$ synchronized.*

Together with Remark 5.6, this proposition allows to test in polynomial time whether a regular set of editing scripts is synchronized.

Remark 5.7. *When L is a regular set of editing scripts given by an automaton, one can compute an automaton for the set $\{t \in L \mid t$ is k-synchronized $\}$. The construction is polynomial in k and L. We will denote this set by $Sync(k, L)$.*

Proposition 5.25. *Fix $k \in \mathbb{N}$. Given a regular set L of editing scripts, $[Sync(k, L)]$ is a regular set of k-synchronized editing scripts.*

Proof. This can be proved using classical constructions on automata. The core of the proof is the storage of a limited information between two siblings labeled with a relabeling. This additional information corresponds to the insertions that are realized between the relabelings. Let $k \in \mathbb{N}$. Let $\mathcal{A} = (\Sigma_{\text{edit}}, Q, \Gamma, I, F, \Delta)$ an automaton over Σ_{edit} such that $L(\mathcal{A})$ is k-insertion-bounded. When $|Q| = 1$, the proposition is trivial so we assume $|Q| > 1$ in the following construction, which simplifies the complexity analysis. We define an automaton $\mathcal{B} = (\Sigma_{\text{edit}}, Q_B, \Gamma_B, I_B, F_B, \Delta_B)$ as follows. The set of states is $Q_B = (Q^2)^{\leqslant k} \times Q \times \{0, \dots, k\} \times \{0, \dots, k\} \cup Q \times \{\circlearrowleft\}$. Clearly, the symbol $\circlearrowleft$ carries no information in the states of Q_B (and likewise for Q_C in the proof of Proposition 5.27). We use the symbol for the sole purpose of identifying a particular kind of states: the states in $Q \times \{\circlearrowleft\}$ can only appear below an insertion or deletion node and mimicks the behaviour of $\mathcal{A}$ on hedges over $\{\varepsilon\} \times \Sigma$ or $\Sigma \times \{\varepsilon\}$. The semantics for the other states is a little more intricate. The first component $(Q^2)^{\leqslant k}$ stores the sequence of subtrees that are presumably inserted between the preceding and following siblings of the current node that are labeled with relabeling nodes. The inserted subtrees are specified as a pair of states, and so the first component stores a word u of length at most k over alphabet Q^2. The unique α such that $u \in (Q^2)^{\alpha}$ is the length of u, denoted by $|u|$, and the i^{th} pair of u is denoted by $u[i]$ for all $i \leqslant |u|$. The second component of the state belongs to Q and basically processes the transitions as in $\mathcal{A}$. The third component records how many subtrees have actually been inserted, whereas the last component records how many insertions have been proved "feasible". The stack alphabet is $\Gamma_B = \Gamma \times (Q^2)^{\leqslant k} \times (Q \cup \bigstar) \times \{0,\ldots,k\} \times \{0,\ldots,k\} \cup \Gamma \times \{\circlearrowleft\}$. Again, states of the form $\Gamma \times \{\circlearrowleft\}$ mimick the behaviour of $\mathcal{A}$. In the other kind of states, the first component Γ basically processes the transitions as in $\mathcal{A}$, and the remaining components are essentially used to propagate the state of the automaton. The initial and final states are $I_B = \{\varepsilon\} \times I \times \{0\} \times \{0\}$ and $\{\varepsilon\} \times F \times \{0\} \times \{0\}$, respectively. The transitions are specified by the following rules:

- For every $a, b \in \Sigma, \eta \in \{op, cl\}$, $i \leqslant k$, $u \in (Q^2)^i$, $u' \in (Q^2)^{\leqslant k}$, and for every transition $q \xrightarrow{(\eta,(a,b)):\gamma} q' \in \Delta$, we add $(u, q, i, i) \xrightarrow{(\eta,(a,b)):(\gamma,\varepsilon,\bigstar,0,0)} (u', q', 0, 0)$ to Δ_B.

- For every $u \in (Q^2)^{\leqslant k}$, $i \leqslant |u|$, and $j < |u|$ such that $u[j+1] = (q, q')$, we add transition $(u, q, i, j) \xrightarrow{\varepsilon} (u, q', i, j+1)$ to Δ_B.

- For every $u \in (Q^2)^{\leqslant k}$, $i, j \in \{0, \ldots, |u|\}$ and every transition $q \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q' \in \Delta$, we add transition $(u, q, i, j) \xrightarrow{(\mathrm{op},(a,\varepsilon)):(\gamma,u,\bigstar,i,j)} (q', \circlearrowleft)$ to Δ_B.

- For every $u \in (Q^2)^{\leqslant k}$, $i, j \leqslant |u|$ and every transition $q \xrightarrow{(\mathrm{cl},(a,\varepsilon)):\gamma} q' \in \Delta$, we add transition $(q, \circlearrowleft) \xrightarrow{(\mathrm{cl},(a,\varepsilon)):(\gamma,u,\bigstar,i,j)} (u, q', i, j)$ to Δ_B.

- For every $u \in (Q^2)^{\leqslant k}$, $i < |u|$, $j \leqslant |u|$, every $p \in Q$, and every pair of transitions $q \xrightarrow{(\mathrm{op},(\varepsilon,a)):\gamma} q'$ and $q'' \xrightarrow{(\mathrm{cl},(\varepsilon,a)):\gamma} q'''$ in Δ such that $u[i+1] = (q, q''')$, we add the transition $(u, p, i, j) \xrightarrow{(\mathrm{op},(\varepsilon,a)):(\gamma,u,p,i,j)} (q', \circlearrowleft)$ to Δ_B. We also add $(q'', \circlearrowleft) \xrightarrow{(\mathrm{cl},(\varepsilon,a)):(\gamma,u,p,i,j)} (u, p, i+1, j)$ to Δ_B.

- For every $\eta \in \{op, cl\}$ and every transition $q \xrightarrow{(\eta,(\varepsilon,a)):\gamma} q'$ in Δ, we add transition $(q, \circlearrowleft) \xrightarrow{(\eta,(\varepsilon,a)):\gamma} (q', \circlearrowleft)$ to Δ_B.

- For every $\eta \in \{op, cl\}$ and every transition $q \xrightarrow{(\eta,(a,\varepsilon)):\gamma} q'$ in Δ, we add transition $(q, \circlearrowleft) \xrightarrow{(\eta,(a,\varepsilon)):\gamma} (q', \circlearrowleft)$ to Δ_B.

Invariant: *Let $q, q' \in Q$, $i, j \leqslant k$, $u \in (Q^2)^{\leqslant k}$, and w a word over $\{op, cl\} \times (\Sigma_{edit} \backslash \Sigma \times \Sigma)$. The VPA $\mathcal{B}$ can reach state (q', u, i, j) after reading w from $(q, u, 0, 0)$ if and only if w admits a decomposition of the form $u_1 v_1 u_2 \ldots v_i u_{i+1}$ such that the following conditions are satisfied:*

1. *$i, j \leqslant |u|$*

2. *each u_l $(l \leqslant i+1)$ is a well nested word over $\Sigma \times \{\varepsilon\}$,*

3. *each v_l $(l \leqslant i)$ is the linearization of some tree over $\{\varepsilon\} \times \Sigma$*

4. *$v_l \in \mathcal{A}_{q_l,q'_l}$ where $u[l] = (q_l, q'_l)$, $(l \leqslant i)$*

5. *there exists a decomposition of the word $u_1u_2 \dots u_{i+1}$ into $j+1$ nested words $x_1, \dots, x_{j+1}$, and there exist j trees $t_1, \dots, t_j$ over $\{\varepsilon\} \times \Sigma$ such that the following two conditions are satisfied: (a) for every $j' \leqslant j$, t_j is accepted by $\mathcal{A}_{s,t}$, where $u[j'] = (s,t)$, and (b) $x_1 lin(t_1) x_2 lin(t_2) \dots x_{i+1}$ is accepted by $\mathcal{A}_{q,q'}$.*

Moreover, $\mathcal{B}$ cannot reach from $(q, u, 0, 0)$ a state that is not of the form (q', u, i, j) if w is a well nested word (over $\Sigma_{edit} \backslash \Sigma \times \Sigma$).

We can deduce another invariant from the form of the transitions for symbols in $\{op, cl\} \times \Sigma \times \Sigma$ together with this invariant:

Invariant: *Let $q' \in Q$, $i, j \leqslant k$, $N \in \mathbb{N}$, $\gamma_1, \dots, \gamma_N \in \Gamma$, and w a word over $\{op, cl\} \times \Sigma_{edit}$. The following two statements are equivalent:*

1. *there exist $u \in (Q^2)^{\leqslant k}$, and $\sigma \in \Gamma_B^*$ with $\pi_1(\sigma) = \gamma_1 \dots \gamma_N$ such that VPA $\mathcal{B}$ can reach a configuration $((q', u, |u|, |u|), \sigma)$ after reading w*

2. *there exists $w' \sim w$ such that $\mathcal{A}$ can reach configuration $(q', \gamma_1 \dots \gamma_N)$ after reading w'.*

This guarantees the correction of the construction. $L(\mathcal{B}) = [L]$, and $|\mathcal{B}| = O(|Q|^{O(k)}|\Sigma|)$, which concludes the proof. The VPA $\mathcal{B}$ has $O(k^2|Q|^{2k+1})$ states and $O(k^2|\Gamma||Q|^{2k+2})$ stack symbols. The number of transitions could be slightly lowered if we observe that the useful information in word u from state (q, u, i, j) is contained in the suffix of u after $\min(i, j)$: $u[\min(i, j)]u[\min(i, j)+1] \dots u[|u|]$. One could therefore reduce the number of transitions generated by the first rule, resulting in an overall number of transitions bounded by $O(|\Delta|^2 + k^2|Q|^{2k}(|\Delta| + |Q|^2))$. We stick to the above non-optimal construction because it can be easily adapted in order to prove Proposition 5.27. □

This result is essentially the adaptation for editing scripts of standard results on trace monoids. For instance, given a k-synchronized regular word language L', the regularity of $[L']$ is an immediate consequence of a result by Métivier [Mét88] on free partially abelian monoids.

We could also define a normal form for the document, shifting all deletions to the left and insertions to the right as far as possible for instance. The set of editing scripts resulting from the normalization of a regular set L of k-synchronized editing scripts is regular.

Notation. *Given a regular view V and $k \geqslant 0$, we denote by $\mathcal{U}_V^k$ the set of all editing scripts u_s such that the editing script induced by u_s on view V is k-synchronized.*

We have a result of the same flavour as the above remark:

Proposition 5.26. *When L is a regular set of editing scripts given by an automaton, one can compute an automaton for the set $L \cap \mathcal{U}_V^k$.*

Proof. We first compute an automaton for the set of all t in $V^{-1} \bowtie L \bowtie V$ such that $\pi_{14}(t)$ is k-synchronized. Then we project this regular language on the second and third component, using Proposition 3.18. □

We next introduce the complementation of a set of editing scripts and show how to combine complementation and equivalence for k-synchronized languages of scripts. The complement of a set of scripts L over alphabet Σ consists of all scripts over alphabet Σ except those of L. We denote by $\complement L$ this complement. Clearly, $\complement L$ is regular for any regular set of scripts L. Moreover, complementation preserves closure under equivalence: $[\complement[L]] = \complement[L]$. We also observe that complement preserves closure under equivalence, and that $Sync(k, \complement Sync(k, L)) = Sync(k, \complement L)$.

Proposition 5.27. *Given a regular view V, $k \in \mathbb{N}$, and a regular set L of k-synchronized editing scripts, one can compute an automaton for the set $\complement[L]$ in exponential time.*

Proof. Let $\mathcal{A} = (\Sigma_{\text{edit}}, Q, \Gamma, I, F, \Delta)$ an automaton over Σ_{edit} such that $L(\mathcal{A}) = L$ is k-insertion-bounded. If we complement naïvely (through determinization) the VPA obtained in Proposition 5.25 for $[L]$, we obtain a doubly exponential complexity since in general the determinization of an automaton with $|Q|^k$ states results in a VPA with $2^{|Q|^{2k}}$ states. What is more, attempts at producing "ad-hoc" determinization procedures seem to be doomed, as a deterministic VPA for $\complement[L]$ would have to store a representation of some set of sequences of (up to) k insertions, and there are over $2^{|Q|^k}$ such sets.

Instead of determinization, we describe an ad-hoc construction that builds an unambiguous VPA for $\complement[L]$. The construction essentially extends with additional components the deterministic automaton obtained from $\mathcal{A}$, instead of using the automaton from $[L]$.

The VPA for $\complement[L]$ has states $Q_C = Q_1 \cup Q_2 \cup Q_3$ with $Q_1 = (\mathcal{P}(Q^2))^{\leqslant k} \times \{0, \ldots, k\} \times \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, $Q_2 = \mathcal{P}(Q^2) \times \{\circlearrowright\}$, and $Q_3 = \{q_\#\}$. The stack alphabet is $\Gamma_C = \Gamma_1 \cup \Gamma_1' \cup \Gamma_2 \cup \Gamma_3$ with $\Gamma_1 = (\mathcal{P}(Q^2))^{\leqslant k} \times \{0, \ldots, k\} \times \mathcal{P}(Q^2 \times \{0, \ldots, k\}) \times \Sigma_{\text{edit}}$, $\Gamma_1' = \{\bigstar\} \times \mathcal{P}(Q^2 \times \{0, \ldots, k\}) \times \Sigma_{\text{edit}}$, $\Gamma_2 = \mathcal{P}(Q^2) \times \{\circlearrowright\} \times \Sigma_{\text{edit}}$, and $\Gamma_3 = \Sigma_{\text{edit}}$. The initial and final states are $I_C = \{(\{\varepsilon\}, 0, \{(q, q, 0) \mid q \in I\})\}$. and $F_C = \{q_\#\} \cup \{\varepsilon\} \times \{0\} \times \mathcal{P}((Q \times (Q \backslash F)) \times \{0\})$. The transitions are defined by the rules below. We first define for every $u \in \mathcal{P}((Q^2)^{\leqslant k})$ and $S \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$ the set $\text{Ext}(u, S)$ as the set of all triples (p, q, j) such that $j \leqslant |u|$ and there exist $i \leqslant j$, $q_0, q_1, \ldots q_{j-i}$ in Q satisfying the following two conditions: (1) $(q, q_0, i) \in S$ and (2) for all $h \in \{0, \ldots, j - i - 1\}$, $(q_h, q_{h+1}) \in u[i + h + 1]$.

- For every $a, b \in \Sigma$, $u \in (\mathcal{P}(Q^2))^{\leqslant k}$, and $S \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, let S' denote the set of all triples $(q, q, 0)$ such that there exists $(p, p') \in S$

with $p' \xrightarrow{(\mathrm{op},(a,b)):\gamma} q \in \Delta$. If $S' \neq \varnothing$, we add for every $u' \in (\mathcal{P}(Q^2))^{\leqslant k}$ the transition $(u, |u|, S) \xrightarrow{(\mathrm{op},(a,b)):(\bigstar,S,(a,b))} (u', 0, \mathrm{Ext}(u', 0, S'))$ to Δ_C. Otherwise, we add transition $(u, |u|, S) \xrightarrow{(\mathrm{op},(a,b)):(q_\#,(a,b))} q_\#$ to Δ_C.

- For every $a, b \in \Sigma$, $u \in (\mathcal{P}(Q^2))^{\leqslant k}$, and $S, S_0 \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, let S' denote the set of all triples $(q_1, q_5, 0)$ such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1) $(q_1, q_2) \in S_0$, (2) $q_2 \xrightarrow{(\mathrm{op},(a,b)):\gamma} q_3 \in \Delta$, (3) $(q_3, q_4, |u|) \in S$, and (4) $q_4 \xrightarrow{(\mathrm{cl},(a,b)):\gamma} q_5 \in \Delta$.

  If S' is not empty, we add for every $u' \in (\mathcal{P}(Q^2))^{\leqslant k}$ the transition $(u, |u|, S) \xrightarrow{(\mathrm{cl},(a,b)):(\bigstar,S_0,(a,b))} (u', 0, \mathrm{Ext}(u', S'))$ to Δ_C. Otherwise, we add transition $(u, |u|, S) \xrightarrow{(\mathrm{cl},(a,b)):(\bigstar,S_0,(a,b))} q_\#$ to Δ_C.

- For every $a \in \Sigma$, and $S \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, let S' denote the set of all pairs (q, q) such that there exist q_1, q_2 and γ satisfying $(q_1, q_2) \in S$ and $q_2 \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q \in \Delta$. Then we add for every $u \in (\mathcal{P}(Q^2))^{\leqslant k}$ and $i \in \{0, \ldots, k\}$ the transition $(u, i, S) \xrightarrow{(\mathrm{op},(a,\varepsilon)):(u,i,S,(a,\varepsilon))} (S', \circlearrowleft)$ to Δ_C.

- For every $a \in \Sigma$, $S_0 \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, and $S \in \mathcal{P}(Q^2)$, let S' denote the set of all triples (q_1, q_5, j) such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1) $(q_1, q_2, j) \in S_0$, (2) $q_2 \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q_3 \in \Delta$, (3) $(q_3, q_4) \in S$, and (4) $q_4 \xrightarrow{(\mathrm{cl},(a,\varepsilon)):\gamma} q_5 \in \Delta$.

  If $S' \neq \varnothing$ then we add for every $u \in (\mathcal{P}(Q^2))^{\leqslant k}$ and $i \in \{0, \ldots, k\}$ the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},(a,\varepsilon)):(u,i,S_0,(a,\varepsilon))} \mathrm{Ext}(u, S')$ to Δ_C. Otherwise we add for every such u and i the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},(a,\varepsilon)):(u,i,S_0,(a,\varepsilon))} q_\#$ to Δ_C.

- For every $S \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, let S' denote the set of all pairs (q, q) such that there exist $q_1 \in Q$ and γ satisfying $q_1 \xrightarrow{(\mathrm{op},(\varepsilon,a)):\gamma} q \in \Delta$. If $S' \neq \varnothing$ then for every $u \in (Q^2)^{\leqslant k}$ and $i \in \{0, \ldots, |u|\}$, we add transition $(u, i, S) \xrightarrow{(\mathrm{op},(\varepsilon,a)):(u,i,S,(\varepsilon,a))} (S', \circlearrowleft)$ to Δ_C. Otherwise we add the transition $(u, i, S) \xrightarrow{(\mathrm{op},(\varepsilon,a)):(\varepsilon,a)} q_\#$ to Δ_C.

- For every $S \in \mathcal{P}(Q^2)$, let S' denote the set of all pairs (q_2, q_5) in Q^2 such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1) $q_2 \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q_3 \in \Delta$, (2) $(q_3, q_4) \in S$, and (3) $q_4 \xrightarrow{(\mathrm{cl},(a,\varepsilon)):\gamma} q_5 \in \Delta$. If $S' = \varnothing$ then for every $u \in (Q^2)^{\leqslant k}$, $i \in \{0, \ldots, |u|\}$, and $S_0 \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, we add a transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},(\varepsilon,a)):(u,i,S_0)} q_\#$ to Δ_C. Otherwise, for every $u \in (Q^2)^{\leqslant k}$, $i \in \{0, \ldots, |u|\}$, if $S' = u[i + 1]$ then we add transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},(\varepsilon,a)):(u,i,S_0,(\varepsilon,a))} (u, i + 1, S_0)$ to Δ_C.

- For every $S \in \mathcal{P}(Q^2)$ and $\alpha \in \Sigma \times \{\varepsilon\} \cup \{\varepsilon\}$, let S' denote the set of all pairs (q,q) such that there exist q_1, q_2 and γ satisfying the following two conditions: (1) $(q_1, q_2) \in S$, (2) $q_2 \xrightarrow{(\mathrm{op},\alpha):\gamma} q \in \Delta$. If $S' = \varnothing$ then we add the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{op},\alpha):\alpha} q_\#$ to Δ_C. Otherwise we add the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{op},\alpha):(S,\circlearrowleft,\alpha)} (S', \circlearrowleft)$ to Δ_C.

- For every $S, S_0 \in \mathcal{P}(Q^2)$ and $\alpha \in \Sigma \times \{\varepsilon\} \cup \{\varepsilon\}$, let S' denote the set of all pairs (q,q) such that there exist q_2, q_3, q_4 and γ satisfying the following four conditions: (1) $(q_1, q_2, j) \in S_0$, (2) $q_2 \xrightarrow{(\mathrm{op},(a,\varepsilon)):\gamma} q_3 \in \Delta$, (3) $(q_3, q_4) \in S$, and (4) $q_4 \xrightarrow{(\mathrm{cl},(a,\varepsilon)):\gamma} q_5 \in \Delta$. If $S' = \varnothing$ then we add for every $u \in (\mathcal{P}(Q^2))^{\leqslant k}$ and $i \in \{0, \ldots, k\}$ the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},\alpha):(S_0,\circlearrowleft,\alpha)} q_\#$ to Δ_C. Otherwise we add for every such u and i the transition $(S, \circlearrowleft) \xrightarrow{(\mathrm{cl},\alpha):(S_0,\circlearrowleft,\alpha)} (S', \circlearrowleft)$ to Δ_C.

- for every $\alpha \in \Sigma_{\mathrm{edit}}$ and $\gamma \in \Gamma_C$ such that the last component of γ is α, we add transitions $q_\# \xrightarrow{(\mathrm{op},\alpha):\gamma} q_\#$ and $q_\# \xrightarrow{(\mathrm{cl},\alpha):\gamma} q_\#$ to Δ_C.

The construction satisfies the following invariant:

> **Invariant:** *Let $u \in (Q^2)^{\leqslant k}$, $i \leqslant k$, $S \in \mathcal{P}(Q^2 \times \{0, \ldots, k\})$, and w a word over $\{op, cl\} \times (\Sigma_{edit} \backslash \Sigma \times \Sigma)$. The VPA $\mathcal{B}$ can reach state (u, i, S) after reading w from (u, i, S) if and only if w admits a decomposition of the form $u_1 v_1 u_2 \ldots v_i u_{i+1}$ such that the following conditions are satisfied:*
>
> 1. *$i \leqslant |u|$*
> 2. *each u_l $(l \leqslant i+1)$ is a well nested word over $\Sigma \times \{\varepsilon\}$,*
> 3. *each v_l $(l \leqslant i)$ is the linearization of some tree over $\{\varepsilon\} \times \Sigma$*
> 4. *$u[l] = \{(q, q') \in Q^2 \mid v_i \in \mathcal{A}_{q_l, q'_l}\}$ $(l \leqslant i)$*
> 5. *S is the set of all pairs $((q, q'), j) \in Q^2 \times \{0, \ldots, |u|\}$ such that there exists a decomposition of the word $u_1 u_2 \ldots u_{i+1}$ into $j+1$ nested words $x_1, \ldots, x_{j+1}$, and there exist j trees $t_1, \ldots, t_j$ over $\{\varepsilon\} \times \Sigma$ such that the following two conditions are satisfied: (a) for every $j' \leqslant j$, t_j is accepted by $\mathcal{A}_{s,t}$, where $(s,t) \in u[j']$, and (b) $x_1 lin(t_1) x_2 lin(t_2) \ldots x_{i+1}$ is accepted by $\mathcal{A}_{q,q'}$.*
>
> *Moreover, when w is a well nested word (over $\Sigma_{edit} \backslash \Sigma \times \Sigma$), $\mathcal{B}$ cannot reach from $(q, u, 0, 0)$ a state that is not of the form (q', u, i, j) or $q_\#$.*

From this invariant and the form of the three transition rules introducing state $q_\#$, we can deduce the following invariant. Let w a word over $\{op, cl\} \times \Sigma_{\mathrm{edit}}$ ending with a letter in $\{op, cl\} \times \Sigma \times \Sigma$. Let u the longest well nested suffix of w (possibly ε), and v the prefix of w before u: $w = vu$.

> **Invariant:** *VPA $\mathcal{B}$ can reach state $q_\#$ after reading w if and only if $\mathcal{A}$ has no run on any word equivalent to w. Moreover, VPA $\mathcal{B}$ can reach state $(\varepsilon, 0, S)$ after reading w if and only if S is not empty and is the set of triples $(q, q', 0)$ such that there exist $u' \sim u$ and $v' \sim v$ that satisfy both (1) $u' \in \mathcal{A}_{q,q'}$, and (2) $\mathcal{A}$ can reach q after reading v'.*

This guarantees the correction of the construction. By construction, the VPA $\mathcal{A}_C$ satisfies $L(\mathcal{A}_C) = \mathbb{C}[L]$. Furthermore, it has $O(k \times 2^{2k|Q|^2})$ states and $O(k \times 2^{(2k|Q|^2)})$ stack symbols. We have thus obtained an automaton of exponential size that accepts $\mathbb{C}[L]$. This concludes the proof. □

Uniform Updatability for Synchronized Updates

Proposition 5.28 (Problem 4). *Testing uniform translatability of a synchronized update function f_v w.r.t. V and $\mathcal{U}_s$ is* Exptime*-complete.*

Proof (outline). Let $\mathcal{A}$ an automaton that accepts f_v, and let k denote its size. Then $V \circ f_v$ is a regular set of k-synchronized editing scripts. Therefore, $[V \circ f_v]$ also is, according to Proposition 5.25. Furthermore, if we take $L_2 = \mathit{Sync}(k, \mathcal{U}_s \circ V)$, f_v is uniformly translatable iff $[V \circ f_v] \subseteq [L_2]$. Moreover, one can compute in exponential time an automaton accepting the complement of $[L_2]$ according to Proposition 5.27. Consequently, one can decide in exponential time the emptiness of the intersection between this complement and $[V \circ f_v]$. Uniform translatability is therefore in Exptime for a synchronized update function f_v.

Conversely, uniform translatability is Exptime-hard, in spite of our assumptions of "translatability" for f_v on page 182, because the problem subsumes regular tree languages inclusion: let L regular tree languages over alphabet Σ. We define $\mathcal{U}_f$ and f_v as the identity transformation over the domains of L and L'. Formally, we define $\Sigma' = \{(a, a) \mid a \in \Sigma\}$, $\mathcal{U}_s = \{t \in T_{\Sigma'} \mid \pi_1(t) \in L\}$, $V = T'_\Sigma$, and $f_v = \{t \in T_{\Sigma'} \mid \pi_1(t) \in L'\}$. Clearly, V is a view, f_v is a 0-synchronized update function (satisfying the assumptions on page 182) and f_v is uniformly translatable if and onlt if $L \subseteq L'$. Admittedly, the example looks strange because one should probably assume $\mathcal{U}_s$ to contain at least the identity. But then we could define a copy of alphabet Σ, f_v as the update that relabels every node into its copy: a node a takes label, say, a' after the update for every $a \in \Sigma$. A similar adaptation of $\mathcal{U}_s$ gives a more sensible example for the reduction. □

We can observe that our Exptime-hardness result relies on $\mathcal{U}_s$ and V's being part of the input. The problem is polynomial in terms of f_v, as it suffices to compute $[V \circ f_v]$ and check it has empty intersection with the complement of $[L_2]$.

Proposition 5.29. *When f_v is a regular set of k-synchronized editing scripts and V is a regular view, we can compute an automaton for its propagations.*

Proof. This proposition holds whenever $[f_v]$ is a regular language. By proposition 5.25, this is the case when f_v is k-synchronized. The set of valid propagations is equivalent to $\mathcal{U}_s \cap \pi_{14}(V \bowtie [f_v] \bowtie V^{-1})$ by proposition 5.7, using (5.1)⇔(5.2). Note that in fact this result implies proposition 5.28 □

The following theorem gives a procedure to compute the set of all uniform k-synchronized updates. This would for instance allow to provide the user with a representation of her possible updates. The construction, however, is blatantly inefficient.

Theorem 5.30 (Problem 5). *We can compute an automaton for the set of all uniform k-synchronized view editing scripts.*

This means in particular that when $\mathcal{U}_s \subseteq \mathcal{U}_V^k$, i.e., when the set of authorized editing scripts is such that the editing scripts it induces on the views are k-synchronized, then we can compute all editing scripts. Both versions of the problem are equivalent since we can build in polynomial time from $\mathcal{U}_s$ an automaton that accepts the language $\mathcal{U}_s \subseteq \mathcal{U}_V^k$.

Proof. First of all, we recall that we assume throughout the chapter that view updates are reasonable and so $\pi_1(u_v) \in \pi_2(V)$ for every view update u_v over view V. In accordance with the above remark, we assume w.l.o.g. that $\mathcal{U}_s \subseteq \mathcal{U}_V^k$. We claim that $\mathit{Unif}\,(V,\mathcal{U}_s)$ is equal to $U_1 = [V^{-1} \circ \mathcal{U}_s \circ V] \cap \mathtt{C}[V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V]]$. The result follows easily from this claim. By assumption, $V^{-1} \circ \mathcal{U}_s \circ V$ is k-synchronized, hence $\mathcal{U}_s \circ V$ also is. We can therefore compute an automaton for the language $\mathtt{C}[\mathcal{U}_s \circ V]$. This yields in turn the construction of an automaton for $\mathit{Sync}(k, V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V])$, hence for $\mathit{Sync}(k, \mathtt{C}[\mathit{Sync}(k, V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V])])$. As observed on page 192, for any language L the languages $\mathit{Sync}(k, \mathtt{C}[\mathit{Sync}(k, L)])$ and $\mathit{Sync}(k, \mathtt{C}[L])$ are equal. Furthermore, we assumed $V^{-1} \circ \mathcal{U}_s \circ V$ to be k-synchronized, so that $[V^{-1} \circ \mathcal{U}_s \circ V] \cap \mathtt{C}[V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V]]$ denotes the same language as $[V^{-1} \circ \mathcal{U}_s \circ V] \cap \mathit{Sync}(k, V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V])$. We have thus proved the result.

Let us now prove the claim. We begin with implication $U_1 \subseteq \mathit{Unif}\,(V,\mathcal{U}_s)$. Let $u_v \in U_1$. According to Proposition 5.15, we must prove there exists some $L \subseteq \mathcal{U}_s$ such that $V^{-1} \circ L \circ V \subseteq [u_v]$ and $\pi_1(V \circ u_v) \subseteq \pi_1(L \circ V)$. Let v_1 any script in V such that $v_1 \circ u_v \neq \emptyset$. We have $v_1^{-1} \circ v_1 \circ u_v = u_v \neq \emptyset$, therefore $v_1 \circ u_v$ cannot belong to $\mathtt{C}[\mathcal{U}_s \circ V]$ since by hypothesis u_v belongs to $\mathtt{C}[V^{-1} \circ \mathtt{C}[\mathcal{U}_s \circ V]]$. Hence $v_1 \circ u_v \in [\mathcal{U}_s \circ V]$. Let $u_1 \in \mathcal{U}_s$ and $v_2 \in V$ such that $v_1 \circ u_v \sim u_1 \circ v_2$. We obtain $v_1^{-1} \circ v_1 \circ u_v \sim v_1^{-1} \circ u_1 \circ v_2$, hence $u_v \sim v_1^{-1} \circ u_1 \circ v_2$. If we take for L the set of all scripts u_1 obtained when v_1 ranges the set of script in V such that $v_1 \circ u_v \neq \emptyset$, we obtain a set $L \subseteq \mathcal{U}_s$ such that $V^{-1} \circ L \circ V \subseteq [u_v]$ and $\pi_1(V \circ u_v) \subseteq \pi_1(L \circ V)$, which concludes the proof of the first implication.

Conversely, we prove that $U_1 \supseteq \mathit{Unif}\,(V,\mathcal{U}_s)$. Let $u_v \in \mathit{Unif}\,(V,\mathcal{U}_s)$. There exists some $L \subseteq \mathcal{U}_s$ such that $L \circ V \sim V \circ u_v$. Consequently, $V^{-1} \circ L \circ$

$V \sim V^{-1} \circ V \circ u_v$ according to Proposition 5.5, hence $V^{-1} \circ L \circ V \sim u_v$. Let finally $w \notin [\mathcal{U}_s \circ V]$. Suppose there were some $v_1 \in V$ such that $u_v \sim v_1^{-1} \circ w$. Then we would get $v_1 \circ u_v \sim v_1 \circ v_1^{-1} \circ w$ according to Proposition 5.5, hence $w \sim v_1 \circ u_v$. This however contradicts $V \circ u_v \subseteq [L \circ V] \subseteq [\mathcal{U}_s \circ V]$. We have thus proved that u_v cannot belong to $[V^{-1} \circ \complement[\mathcal{U}_s \circ V]]$, which concludes the proof of the claim.

We have put a thoroughly different construction for the automaton that accepts uniform view editing scripts in the appendix. The construction of the appendix is more "direct" as it avoids the double complementation process. This makes the construction more intuitive and possibly more efficient, although it presents similar worst case doubly exponential complexity. □

We may be interested also in restricting the set of authorized editing scripts. For instance, given a fixed view V, one may wish the set $\mathcal{U}_s$ to be such that every view editing script u_v has a unique propagation from each source document:

Definition 5.12. *A set of source editing scripts $\mathcal{U}_s$ is V-*unambiguous *if for all u_s, u_s' in $\mathcal{U}_s$ such that $\pi_1(u_s) = \pi_1(u_s')$, either $u_s \sim u_s'$ or $u_s \circ V$ and $u_s' \circ V$ are not equivalent.*

Given a regular set of (stable) editing scripts $\mathcal{U}_s$ and a view V, we would like to compute $\mathcal{U}_s' \subseteq [\mathcal{U}_s]$ such that $\mathcal{U}_s' \circ V = \mathcal{U}_s \circ V$ and $\mathcal{U}_s'$ is V-unambiguous. In general, this disambiguation of $\mathcal{U}_s$ cannot be achieved.

Proposition 5.31. *Testing whether $\mathcal{U}_s$ is V-unambiguous is undecidable.*

We can prove similarly there is no algorithm that can compute a (regular) set disambiguating $\mathcal{U}_s$. However, for k-synchronized editing scripts, the disambiguation can be achieved.

Theorem 5.32. *Given a view V, and $\mathcal{U}_s \subseteq \mathcal{U}_V^k$, we can compute a regular set of editing scripts $\mathcal{U}_s' \subseteq \mathcal{U}_s$ such that $\mathcal{U}_s' \circ V = \mathcal{U}_s \circ V$ and $\mathcal{U}_s'$ is V-unambiguous.*

Proof. We proceed by an exponential reduction to Theorem 5.14. Given an editing scripts t, the *left normal form* of t is the unique script $\mathrm{lnf}(t)$ equivalent to t such that for every nodes $n.i, n.(i+1) \in N_t$, $lab_t(n.i) \in \Sigma \times \{\varepsilon\} \implies lab_t(n.(i+1)) \notin \{\varepsilon\} \times \Sigma$. Intuitively, it is obtained from t by placing insertions before deletions for the following-sibling ordering. As usual, this notion is extended to sets of alignments: $\mathrm{lnf}(L) = \{\mathrm{lnf}(t) \mid t \in L\}$. Fix a view V, and $\mathcal{U}_s \subseteq \mathcal{U}_V^k$. Let $L_0 = \mathrm{lnf}(\mathcal{U}_s \circ V)$. This is a regular set of editing scripts since we supposed $\mathcal{U}_s \subseteq \mathcal{U}_V^k$. Therefore, we can compute an automaton for the set of alignments: $L = \{t \in \Sigma_{\mathrm{edit},3} \mid \pi_{1,3}(t) \in L_0 \wedge \pi_{2,3}(t) \in T_{\Sigma'}\}$, where $\Sigma' = \{(\alpha, \alpha) \mid \alpha \in \Sigma_\varepsilon\}$. We consider L as an editing script with first component over alphabet Σ_{edit} and second component over alphabet Σ_{ε}. Then, applying Theorem 5.14, we can compute a regular language L' such that $L' \subseteq L$ and the domains of L' and L are equal. Posing $\mathcal{U}'_s = \pi_{1,3}(L')$, we get: $\mathcal{U}'_s \circ V = \mathcal{U}_s \circ V$ and $\mathcal{U}'_s$ is V-unambiguous. □

To conclude, let us add a few words about our definition for equivalence. Not only does this definition take better account of identifiers and data-values, it also helps incidentally to define the k-synchronized restriction. If we had defined the equivalence as the equality of the relations, i.e., letting t and t' be equivalent if and only if $\pi_1(t) = \pi_1(t')$ and $\pi_2(t) = \pi_2(t')$, we could have adapted most of our results except for this last local restriction on the number of insertions defining which we called "k-synchronized updates".

6. The View Schema

Contents

The non-materialized framework relies on the presentation of a view schema to the user, so that she can formulate her queries. Ideally, for a view V over domain D_0, the view schema should be a representation of $\mathcal{V}iew\,(V, D_0)$, a DTD representation for instance. In the most general case D_0 is an arbitrary regular language, and V is an arbitrary view (i.e., a regular set of alignments). We also consider the effect of various restrictions on D_0 and V on the view schema.

Two different obstacles may arise when we try to compute a DTD D satisfying $L(D) = \mathcal{V}iew\,(V, D_0)$. On the one hand, there may be no DTD that represents the language $\mathcal{V}iew\,(V, D_0)$. Representing the view schema indeed raises the question of the expressivity of our views: views that are too expressive yield view schemata that need not even be regular. But on the other hand, even when the view schema can be represented with an XML DTD, the computation of this view schema might prove intractable, and the size of the resulting schema may be prohibitive. This raises the question of the complexity for computing a view schema.

The first section of this chapter surveys the expressivity of various restrictions on the view. We then investigate the particular case of XML DTDs where the regular expressions are required to be deterministic. Finally, we provide a few algorithms to approximate the view schema when needed. We have no quick fix for the complexity problem, though: only the most simple of our approximations is guaranteed to give a small and easily computable view schema.

6.1. Computing the View Schema

First of all, $\mathcal{V}iew\,(V, D_0)$ need not be a local tree language, as illustrated by (D_1, ann_1) in Example 6.1. When $\mathcal{V}iew\,(V, D_0)$ is a regular language, we can resort to EDTDs, or similar representations that overcome the limitations of DTDs. However $\mathcal{V}iew\,(V, D_0)$ need not always be regular, when internal nodes are deleted as in (D_2, ann_2) from Example 6.1. In any case, (D_3, ann_3) shows that the deletion of internal nodes may result in an exponential blowup in the size of the view schema.

Example 6.1. *We consider the three annotated DTDs* (D_1, ann_1), (D_2, ann_2), *and* (D_3, ann_3) *specified below. Let* Q_1, Q_2 *and* Q_3 *denote queries representing those annotated DTDs.*

$$\begin{array}{lll}
D_1 : \mathtt{r} \to \mathsf{abc} & D_2 : \mathtt{r} \to \mathsf{c} & D_3 : \mathtt{r} \to \mathsf{a}_k \\
\quad\;\; \mathsf{b} \to \mathsf{d} \mid \mathsf{e} & \quad\;\; \mathsf{c} \to (\mathsf{acb}) \mid \varepsilon & \quad\;\; \mathsf{a}_i \to \mathsf{a}_{i-1}\mathsf{a}_{i-1} \\
 & & \quad\;\; \mathsf{a}_0 \to \mathsf{a} \\
\mathsf{ann}_1(\mathtt{r}, \mathsf{a}) = [\Rightarrow/\Downarrow{::}\mathsf{d}], & \mathsf{ann}_2(\mathtt{r}, \mathsf{c}) = \mathsf{ann}_2(\mathsf{c}, \mathsf{c}) = \mathsf{false}, & \mathsf{ann}_3(\mathtt{r}, \mathsf{a}_k) = \mathsf{false}, \\
\mathsf{ann}_1(\mathtt{r}, \mathsf{c}) = [\Leftarrow/\Downarrow{::}\mathsf{d}], & \mathsf{ann}_2(\mathsf{c}, \mathsf{a}) = \mathsf{ann}_2(\mathsf{c}, \mathsf{b}) = \mathsf{true}, & \mathsf{ann}_3(\mathsf{a}_0, \mathsf{a}) = \mathsf{true}.
\end{array}$$

We observe that $\mathcal{V}iew\,(Q_1, D_1) = \{\mathtt{r}(\mathsf{a}, \mathsf{b}(\mathsf{d}), \mathsf{c}), \mathtt{r}(\mathsf{b}(\mathsf{e}))\}$, *although regular, cannot be captured with a DTD, whereas* $\mathcal{V}iew\,(Q_2, D_2) = \{\mathtt{r}(\mathsf{a}^k\mathsf{b}^k) \mid k \in \mathbb{N}\}$ *is not even a regular tree language. And finally,* $\mathcal{V}iew\,(Q_3, D_3) = \{\mathtt{r}(\mathsf{a}^{2^k})\}$ *can be captured by a DTD but requires a DTD of size exponential in* $|D_3|$.

For simple and general ($\mathcal{X}Reg$) annotations, we survey the impact of the interval-boundedness and upward-closure assumptions on the expressiveness of the view schema.

Simple Annotations When the view is specified by an annotated DTD (D, ann) where ann maps each pair into true or false, i.e., for simple annotations, the view schema is local. For upward-closed or even for interval-bounded annotations, the view schema is regular. Consequently, the view schema can be represented by a DTD for every interval-bounded simple annotation. Reciprocally, any DTD is trivially the view schema of some upward-closed simple annotation if we consider an annotation that does not hide any node. Without restrictions, however, simple annotations do not guarantee a regular view schema, as illustrated by D_2. But the view schema can always be expressed with a CDTD. Reciprocally, every CDTD $D_0 = (\Sigma, r, P)$ is the view schema of some simple annotation: take for D a DTD whose symbols are the union of Σ and of all terminals and non terminals occurring in some grammar from P. We can suppose without loss of generality that all grammars in P use distinct non-terminals. The production rules of D are then defined from those in the grammars from P, and the annotation hides the non-terminals.

Regular XPath Annotations The view schema obtained for any interval-bounded Regular XPath annotation or even for any interval-bounded *MSO* query is regular according to Proposition 4.2, and thus can be expressed as an EDTD. Reciprocally, every EDTD is the view of some DTD for some $\mathcal{X}Reg$ annotation. The proof of this is quite similar to the proof for Proposition 4.37 and can be immediately deduced from the one for Lemma 6.2:

Lemma 6.1. *Let $\mathcal{E} = (\Sigma, \Sigma', D, \mu)$ an arbitrary EDTD. There exists an interval bounded $\mathcal{X}Reg$ query, satisfying $\mathcal{V}iew\,(Q, T_\Sigma) = L(\mathcal{E})$.*

Proof. The result follows immediately from the proof of Lemma 6.2: when $\mathcal{E}$ is an EDTD, the productions are given immediately with a regular expression, there is no need for non-terminals except the start symbol, and therefore the query Q is 1-interval bounded. □

Using views that are not interval bounded brings hassle when it comes to constructing the view schema: the view schema may have to represent an arbitrary context-free language (if we consider view trees of depth one). We prove that every ECDTD can be expressed as the view of some $\mathcal{X}Reg$ query.

Lemma 6.2. *Let $\mathcal{E} = (\Sigma, \Sigma', D, \mu)$ an arbitrary ECDTD. There exists a general $\mathcal{X}Reg$ query, satisfying $\mathcal{V}iew\,(Q, T_\Sigma) = L(\mathcal{E})$.*

Proof. Let $D = (\Sigma', r, P)$ the underlying DTD in $\mathcal{E}$. We define a $\mathcal{X}Reg$ query Q and DTD D_0 such that $\mathcal{V}iew\,(Q, D_0) = L(\mathcal{E})$. The proposition follows immediately by Lemma 3.17. For technical reasons, we assume that $\mathcal{E}$ does not relabel its root, that Σ and Σ' have only the root symbol r in common, and that the single node tree r consisting of the root only belongs to $L(\mathcal{E})$. Those assumptions could be disposed of at the cost of "heavier" constructions.

We further assume that non-terminals appearing in the context-free grammars $P(a)$ and $P(b)$ are disjoint for every $a \neq b \in \Sigma'$, and are also disjoint from Σ and Σ'. We denote the union of all such non-terminals by $\mathcal{V}$. We define a new DTD $D_0 = (\Sigma \cup \Sigma' \cup \mathcal{V}, r, R_0)$, where P_0 is defined as follows. Let $\alpha \in \Sigma$. We denote the symbols from Σ' satisfying $\mu(\beta) = \alpha$ by $\beta_1, \ldots, \beta_k$, and denote by $S_1, \ldots, S_k$ the start symbols from $P(\beta_1), \ldots, P(\beta_k)$. Then we define $R_0(\alpha)$ as $S_1 + \cdots + S_k$. For every $U \in \mathcal{V}$, U appears in a single grammar $G_i = (V, T, S_i, P_i)$ from P. We define $R_0(U)$ as the regular expression obtained from $P_i(U)$ by replacing every symbol $\beta \in \Sigma'$ by the two symbols $\beta\mu(\beta)$. $R_0(\beta)$ is set to ε for every $\beta \in \Sigma'$.

One can easily build a query Q that checks, for every node with label in Σ whose left sibling has label β, that its child is labeled with the initial symbol in grammar $P(\beta)$. This query selects only the root r if the verification fails, and select every node with label in Σ otherwise. Such a query will satisfy $\mathcal{V}iew\,(Q, D_0) = L(\mathcal{E})$. □

Reciprocally, the view of a $\mathcal{X}Reg$ query or MSO query can always be expressed as an ECDTD. We can easily prove that $\mathcal{V}iew\,(Q, D)$ is the language of some ECDTD for every regular tree language D and every query Q in $\mathcal{X}Reg$ or even MSO, by encoding the state of the automaton into the labels from Σ' but not from Σ. This concludes the characterization of the view languages obtained for $\mathcal{X}Reg$ queries.

We observe that assuming the source schema to be an XML DTD has no effect on the above results. For instance, every EDTD is the view of some XML DTD for some $\mathcal{X}Reg$ annotation. We summarize the results in Figure 6.1 in the case when the view is defined with an annotated DTD. The table still holds if we suppose the domain to be given by an XML DTD instead of an arbitrary DTD. Also, the first line of the table remains the same if we use query automata instead of $\mathcal{X}Reg$ annotations to define the views.

	general	interval bounded	upward-closed
$\mathcal{X}Reg$ annotation	ECDTD	EDTD	EDTD
simple annotation	CDTD	DTD	DTD

Figure 6.1.: View language obtained by each annotation when the domain is a DTD (same table from an XML DTD).

Testing if The View Schema can be Expressed as a DTD/EDTD... Given an ECDTD or a CDTD, we cannot decide if there exists an EDTD (or a DTD, or even an XML DTD) with the same language. This follows immediately from the undecidability of $Memb(\text{CFL}, MSO)$, the problem of testing if the language of a context-free grammar is regular, which remains undecidable for deterministic regular languages according to Proposition 3.4. However, given an EDTD, we can decide if there exists some equivalent DTD, and this problem is Exptime-complete [MNSB06] as we already mentioned on page 86. Moreover, given a DTD, we can decide (1) if it is an XML DTD, in polynomial (linear) time, and if not (2) if there exists some equivalent XML DTD, in exponential time using a result from Brüggeman-Klein and Wood, as we shall discuss in the next section. Bex et al. [BGMN09] show the problem is Pspace-hard, and leave it as an open question whether it is Pspace-complete. Combining these results, testing if an EDTD admits an equivalent XML DTD is Exptime-complete: the lower bound carries over (for instance considering NTAs over binary trees), and the upper bound is obtained by first testing if the EDTD admits an equivalent DTD in exponential time, in which case the DTD is of linear size, and we can test if the regular expressions occurring in the DTD denote deterministic languages in exponential time.

In the following sections, we address the problems of computing a representation or an approximation by a DTD (or an XML DTD) for the view schema. In the first section, we investigate the verification of determinism in XML DTD. More exactly, we discuss how determinism of regular expressions can be checked, and propose a few solutions when a DTD fails the verification for determinism. Also, we survey some algorithmic issues associated with XML DTDs and discuss, among others, how they allow for faster validation of a file against the schema. We finally outline some solutions to approximate the view schema when obtaining a DTD (or an XML DTD) representing the view language proves unfeasible.

6.2. Determinism in View Schema: XML DTDs

In a favorable setting, a DTD for the view schema may be proposed by the administrator. In this case, checking if this DTD is an XML DTD is an easy task: for each production rule, we check if the regular expression is deterministic. Given a regular expression, Brüggeman-Klein proposes a quadratic algorithm [1] in $O(|\Sigma| \times |e|)$ for checking the determinism of a regular expression e over alphabet Σ. This algorithm simply computes the Glushkov automaton for e and checks if this automaton is deterministic. We improve the complexity into a truly linear algorithm in the next section:

Theorem 6.3. *Determinism of a regular expression e can be tested in time $O(|e|)$, for an arbitrary alphabet.*

This algorithm assumes a RAM model with word size logarithmic in e. We also discuss how our linear time algorithm can be combined with the method from Kilpelaïnen and Tuhkanen [KT07] to obtain a linear algorithm for testing determinism of a regular expression with numeric occurrence, which finds applications in the validation of XML Schema documents [KT07, Kil11].

6.2.1. Linear Algorithm to Test Determinism

We investigate in this section the problem of testing determinism:

Problem 6. *Given a regular expression e, test if e is deterministic.*

We show that determinism can be tested in $O(|e|)$ linear time, thus improving upon the $O(|e| \times |\Sigma|)$ quadratic algorithm by Brüggeman-Klein and Wood [BKW98] when the size of the alphabet is not bounded by a constant. The alphabet Σ remains a finite set of symbol, but its size may be of the

[1] She actually claims a linear algorithm, but she assumes a constant size alphabet

same order than $|e|$. This is actually often the case, since single occurrence regular expressions form a huge majority of the expressions that occurr in practice according to [BNSV10].

We provide two different algorithms. One proceeds by reduction to the evaluation of a Regular XPath formula with data equality, using the property that such a formula can be evaluated with linear data complexity [BP11]. The other is an ad-hoc algorithm. The constructions underlying this latter algorithm have been extended in [GMS12] to provide new algorithms for the evaluation of deterministic regular expressions.

Our algorithm for testing determinism does not construct the Glushkov automaton of e, but works directly on the parse tree of e. Similarly to the construction of the Glushkov automaton, it exploits the relations *Follow*, *First* and *Last* as defined in Section 3.1.3. We cannot compute these relations for every subexpression of e but we provide algorithms to check efficiently if an element belongs to these relations.

We slightly modify the definition of regular expressions from Section 3.1.3. First of all, we assume that every expression accepts at least one word of size one or more, otherwise the expression is obviously deterministic. Then we do not use ε in our syntax for expressions, but use symbol "?" instead. In this section, *regular expressions over* Σ are thus defined by the following grammar, where $\odot$ represents concatenation, $+$ union, ? choice, and $*$ the Kleene star: $e \coloneqq a\ (a \in \Sigma) \mid (e) \odot (e) \mid (e) + (e) \mid (e)? \mid (e)^*$. Note that $L((e)?) = L(e) \cup \{\varepsilon\}$, where ε denotes the empty word. In expressions, we do not write parentheses around words over Σ and often omit $\odot$ symbols. We require of our regular expressions e that:

(R1) $e = (\#e')\$$ and $\#$ and $\$$ do not appear in e'

(R2) $((e')^*)^*$ does not appear in e

(R3) if $(e')?$ appears in e, then $\varepsilon \notin L(e')$

An arbitrary regular expression can be changed easily (in linear time) into an equivalent one of the required form.

We identify a regular expression with its parse tree (as illustrated in Figure 6.2), and define the *positions* $Pos(e)$ of e as the leaves of e whereas N_e denotes the set of all nodes from e. For a node $n \in N_e$ we denote by e/n the subexpression of e rooted at n. Every tree t is implemented as a pointer structure, where $Lchild_t(n)$ (resp. $Rchild_t(n)$) returns the left (resp. right) child of node n in t and $parent_t(n)$ returns the parent of n in t. The pointers return *Null* if the respective node does not exist. For unary nodes $Rchild_t(n)$ returns *Null*. We denote by $lab_t(n)$ the label of n in t, and by $\preccurlyeq_t$ the (reflexive) ancestor relationship in t. If $m \preccurlyeq_t n$ then we also say that n is a descendant of m. Thus, each node is ancestor and descendant of itself.

The size of e is as defined on page 52, with $|(e)?| = 1 + |e|$. Consequently, the size of an expression corresponds to the size of its parse tree. Furthermore, restrictions (R2) and (R3) guarantee that $|e|$ is linear in $|Pos(e)|$.

We say that e is *nullable* if $\varepsilon \in L(e)$. Nullability can be expressed inductively, as already observed in the literature (see, e.g., [BK93]): $e_1 \odot e_2$ is nullable if and only if both e_1 and e_2 are nullable, $e_1 + e_2$ is nullable if and only if e_1 or e_2 is nullable, e_1^* and $e_1?$ are always nullable, whereas $a \in \Sigma$ is never nullable.

Whenever the regular expression or the tree is clear from context, we drop the subscript and write *Follow*, *lab*, and $\preccurlyeq$. We call a pair of positions $q, q' \in Pos(e)$ a *witness for non-determinism* if it satisfies the following two conditions: (1) $lab_e(q) = lab_e(q')$ and (2) there exists some $p \in Pos(t)$ such that $q, q' \in Follow_e(p)$. Thanks to assumption (R1), we can rephrase the traditional definition of determinism from section 3.1.3 as follows: an expression is non deterministic if it admits a witness for non-determinism, otherwise it is deterministic.

Example 6.2. *Let* $e_1 = (ab + b(b?)a)^*$ *and* $e_2 = (a^*ba + bb)^*$. *Denote by* $p_1, \ldots, p_5$ *the positions of* e_1 *in left-to-right order, and by* $q_1, \ldots, q_5$ *those of* e_2. *Then* $\overline{e_1} = (a_1b_2 + b_3(b_4?)a_5)^*$ *and* $Follow_{e_1}(p_3) = \{p_4, p_5\}$. *Similarly,* $\overline{e_2} = (a_1^*b_2a_3 + b_4b_5)^*$, *and* $Follow_{e_2}(q_3) = \{q_1, q_2, q_4\}$. *Expression* e_1 *is deterministic, while* e_2 *is non-deterministic because* $lab_{e_2}(q_2) = lab_{e_2}(q_4) = b$.

To conclude these preliminaries, we introduce lowest common ancestors queries, which are the cornerstone of our algorithm. Given a tree t and two nodes n and n' in t, we denote by $LCA_t(n, n')$ the lowest common ancestor of n and n' in t. As usual, we drop the subscript when it can be deduced from context. Harel and Tarjan [HT84] proved that after a linear preprocessing, one can answer in constant time lowest common ancestor queries. The constants have been improved in a subsequent series of papers, and Bender et al. [BFCP$^+$05] evaluate the performance of the algorithm experimentally: they observe that the algorithm does not significantly outperform (and is even often outperformed by) algorithms with slightly higher asymptotic complexity. Alstrup et al. [AGKR04] survey most of those constructions and proposes a labeling scheme for nearest common ancestor queries, whereas the other constructions require external data structures such as a few arrays.

Lemma 6.4 ([HT84]). *A tree* t *can be preprocessed in time* $O(|t|)$, *so that for every nodes* u *and* v *in* t, $LCA(u, v)$ *can be computed in constant time.*

Similarly, one can preprocess a tree t in linear time to answer ancestor queries in constant time. One possible solution to the problem is to use the LCA preprocessing, as $u \preccurlyeq v$ if and only if $LCA(u, v) = u$. A simpler technique to answer such queries in constant time could be the use of rich identifiers, storing for instance the pre- and post-order numbers of the node, that is, the position of its opening and closing tags in the linearization. Smarter labeling scheme have actually been devised to minimize the size of the labels required for ancestor queries, but the complexity remains essentially the same.

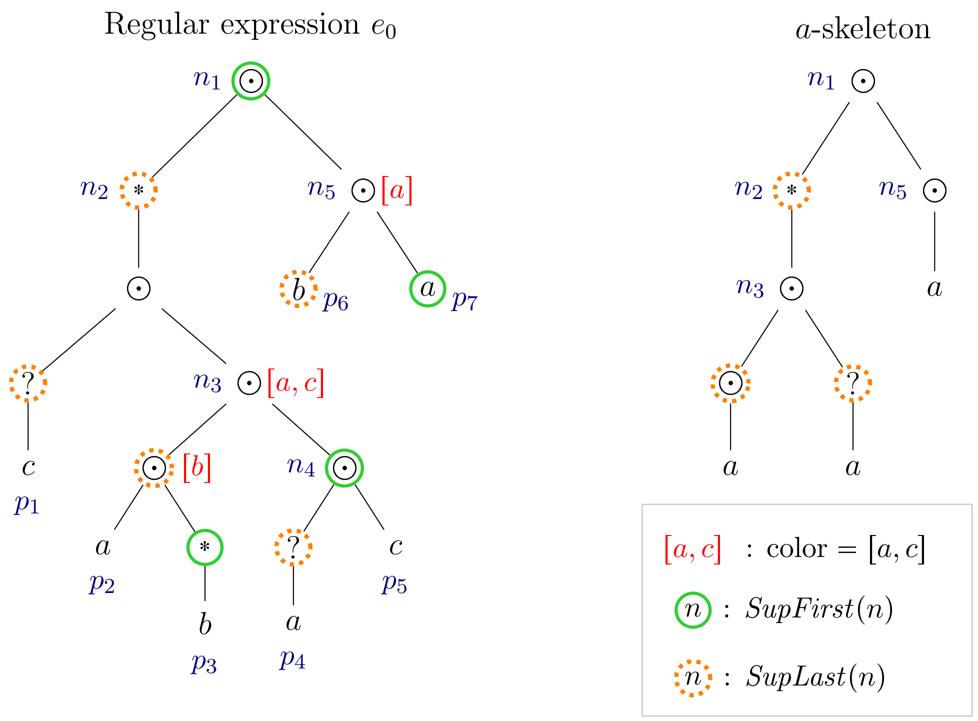


Figure 6.2.: Expression $e_0 = (c?((ab^*)(a?c)))^*(ba)$.

Remark 6.1. *After a linear preprocessing of t, one can decide in constant time for any nodes $u, v \in N_t$ if $u \preccurlyeq_t v$.*

Structural properties of the expression

We first survey some properties of the relations *First*, *Last* and *Follow*. The next lemma was stated before, e.g., in [CP97, PZC96], but not explicitly in terms of lowest common ancestors.

Lemma 6.5. *Let $p, q \in Pos(e)$ and $n = LCA(p, q)$. Then $q \in Follow(p)$ iff one of the following two conditions is satisfied:*

(1) $lab(n) = \odot$, $q \in First(Rchild(n))$, and $p \in Last(Lchild(n))$, or

(2) n has some ancestor labeled with $*$ and, if we denote by s the lowest $*$-labeled ancestor of n, then $q \in First(s)$ and $p \in Last(s)$.

Lemma 6.5 says that there are only two ways in which positions follow each other: (1) through a concatenation, or (2) through a star. We write $q \in Follow_e^{\odot}(p)$ if (1) is satisfied, and $q \in Follow_e^{*}(p)$ if (2) is satisfied. For instance, in e_0 (Figure 6.2), we have $p_4 \in Follow_{e_0}^{\odot}(p_3)$ and $p_1 \in Follow_{e_0}^{*}(p_5)$. Note, however, that there may exist some positions p and q that satisfy simultaneously (1) and (2).

$$\begin{array}{lcl}
\mathit{First}(p) & = & p \text{ for all } p \in \mathit{Pos}(e)\text{, i.e., if } \mathit{lab}(p) \in \Sigma \\
\mathit{First}(e_1 \odot e_2) & = & \begin{cases} \mathit{First}(e_1) \cup \mathit{First}(e_2) & \text{if } e_1 \text{ is nullable} \\ \mathit{First}(e_1) & \text{otherwise} \end{cases} \\
\mathit{First}(e_1 + e_2) & = & \mathit{First}(e_1) \cup \mathit{First}(e_2) \\
\mathit{First}(e_1^*) & = & \mathit{First}(e_1) \\
\mathit{First}(e_1?) & = & \mathit{First}(e_1)
\end{array}$$

$$\begin{array}{lcl}
\mathit{Last}(p) & = & p \text{ for all } p \in \mathit{Pos}(e)\text{, i.e., if } \mathit{lab}(p) \in \Sigma \\
\mathit{Last}(e_1 \odot e_2) & = & \begin{cases} \mathit{Last}(e_1) \cup \mathit{Last}(e_2) & \text{if } e_2 \text{ is nullable} \\ \mathit{Last}(e_2) & \text{otherwise} \end{cases} \\
\mathit{Last}(e_1 + e_2) & = & \mathit{Last}(e_1) \cup \mathit{Last}(e_2) \\
\mathit{Last}(e_1^*) & = & \mathit{Last}(e_1) \\
\mathit{Last}(e_1?) & = & \mathit{Last}(e_1)
\end{array}$$

Figure 6.3.: Inductive definition for the *First* and *Last* sets.

It was also observed earlier, e.g., [CP97, PZC96, HM98], that First and Last-sets (and nullability) can be defined in a syntax-directed way over the parse tree of e. Figure 6.3 summarizes this inductive definition of the first and last sets.

We observe on Figure 6.3 that for every node n, either $\mathit{First}(\mathit{parent}(n))$ contains $\mathit{First}(n)$ or they are disjoint, and in the latter case $\mathit{First}(n)$ is also disjoint with $\mathit{First}(n')$ for every ancestor n' of n. The same property also holds for the *Last* sets. On the basis of this observation, we now define two Boolean properties *SupFirst* and *SupLast* for every node n, where n' denotes the parent of n:

$$\mathit{SupFirst}(n) \text{ iff } \mathit{lab}(n') = \odot, n = \mathit{Rchild}(n'), \text{ and } \mathit{Lchild}(n') \text{ is non-nullable.}$$

$$\mathit{SupLast}(n) \text{ iff } \mathit{lab}(n') = \odot, n = \mathit{Lchild}(n'), \text{ and } \mathit{Rchild}(n') \text{ is non-nullable.}$$

For every node n, $\mathit{SupFirst}(n)$ holds if and only if the First set of n is disjoint from the one of its parent, and the same holds for *SupLast* and the *Last* set. We define two pointers $\mathit{SupFirst}(n)$ and $\mathit{SupLast}(n)$ for every node n. Pointer $\mathit{SupFirst}(n)$ points to the lowest ancestor x of n such that $\mathit{SupFirst}(x)$. Similarly, $\mathit{SupLast}(n)$ points to the lowest ancestor x of n such that $\mathit{SupLast}(x)$. Recall that by (R1) we assumed e to be of the form $(\#e')\$$; this implies that for every node n in e' the pointers $\mathit{SupLast}(n)$ and $\mathit{SupFirst}(n)$ are well defined. Those pointers will never be applied to the “help nodes” that are not in e'. We also define for every node n a pointer $\mathit{pStar}(n)$ toward the lowest ancestor of n labeled with a Kleene star (possibly n itself). If there is no such ancestor, $\mathit{pStar}(n) = \mathit{Null}$. During preprocessing we compute the pointers *SupFirst* and *SupLast* for every node in e, in linear time.

The relations First and Last can be expressed in terms of ancestorship with respect to the *SupFirst* and *SupLast* pointers:

Lemma 6.6. *Let $p \in Pos(e)$ and $n \in N_e$.*

(1) $p \in First(n)$ iff $SupFirst(p) \preccurlyeq n \preccurlyeq p$, and

(2) $p \in Last(n)$ iff $SupLast(p) \preccurlyeq n \preccurlyeq p$.

The following technical lemmas state relationships between positions and their *SupFirst* and *SupLast* nodes.

Lemma 6.7. *Let $p, q \in Pos(e)$ and $q \in Follow_e(p)$. Then*

(1) $parent(SupFirst(q)) \preccurlyeq p$ and

(2) $parent(SupLast(p)) \preccurlyeq q$.

Proof. To show (1), assume that $parent(SupFirst(q))$ is not an ancestor of p. Then $n = LCA(p,q)$ is an ancestor of $parent(SupFirst(q))$, hence $SupFirst(q) \not\preccurlyeq n$. By Lemma 6.6(1) we obtain $q \notin First(n)$ and therefore, by Lemma 6.5, q does not follow p. Point (2) can be proved similarly. □

Lemma 6.8. *Let p and q be two positions of e such that q follows p. If $SupLast(p) \preccurlyeq parent(SupFirst(q))$ then $SupFirst(q)$ is nullable.*

Proof. Let $p, q \in Pos(e)$ such that $SupLast(p) \preccurlyeq parent(SupFirst(q))$ and $q \in Follow(p)$, and let $x = LCA(p,q)$. Assume first that $q \in Follow^{\odot}(p)$. Then $lab(x) = \odot$ and there are no *SupLast* nodes between p and $SupLast(p)$ except $SupLast(p)$. It means that in particular $Rchild(x)$ is nullable. Hence $SupFirst(q)$ is nullable if it is the right-child of x. Otherwise $SupFirst(q)$ is an ancestor of x. In that case, there are no *SupFirst* nodes between q and $SupFirst(q)$, except $SupFirst(q)$, so that $Lchild(x)$ is nullable. Consequently, x is nullable, and there are no *SupFirst* nor *SupLast* nodes between x and $SupFirst(q)$, except the node $SupFirst(q)$. Therefore, $SupFirst(q)$ is nullable. The case $q \in Follow^{*}(p)$ is handled similarly: $pStar(x)$ is nullable and satisfies $SupFirst(q) \preccurlyeq pStar(x) \preccurlyeq x$. Moreover there are no *SupFirst* nor *SupLast* nodes between x and $SupFirst(q)$, except $SupFirst(q)$. Thus, $SupFirst(q)$ is nullable. □

> **Algorithm scheme:** Our algorithm to test determinism searches a witness for non-determinism (q,q'). We must take care of the quadratic number of candidate pairs (q,q'), and moreover we cannot afford to enumerate all positions p to check if $q,q' \in Follow_e(p)$.

We prove that only a linear number of pairs (q,q') must be considered in order to establish whether e is deterministic or not, and for each pair one can decide in constant time if it is a witness for non-determinism, that is, if there exists a position p followed by both q and q'.

Candidate Pair Reduction

Let P1(e) denote the condition:

> "for every $q \neq q'$ in $Pos(e)$, $SupFirst(q) = SupFirst(q')$ implies $lab(q) \neq lab(q')$".

We claim that every deterministic expression satisfies (P1). Indeed, let e a deterministic expression. Let q and q' two distinct positions of e such that $SupFirst(q) = SupFirst(q')$. We denote by n this node $SupFirst(q)$. Since the First and Last sets of any node are non-empty, there exists some position p in $Last(Lchild(parent(n)))$. By definition, $parent(n) = LCA(p, q) = LCA(p, q')$. By Lemma 6.5, $q, q' \in Follow_e(p)$, hence $lab(q) \neq lab(q')$ by definition of determinism.

Testing (P1) in linear time is straightforward: during one traversal of e we group the positions with same *SupFirst*-pointer. In a second step we check that all positions of a same group have distinct labels, for every group. This can easily be achieved in linear time with a single array of size Σ. Therefore we assume from now on that e satisfies (P1).

Point (1) of Lemma 6.7 suggests to store information about position q in the parent n of $SupFirst(q)$: for every position p, if $q \in Follow(p)$ then n is an ancestor of p. For each position p labeled a, we therefore

- assign *color* a to the node $parent(SupFirst(p))$
- say that p is a *witness* for color a in the node $parent(\ SupFirst(p))$. [2]

Observe that each node may be assigned several colors, but, since (P1) holds, each node has at most one witness per color.

Example 6.3. *In Figure 6.2, node n_3 has colors a and c. The witness for color a (resp. c) in n_3 is p_4 (resp. p_5).*

Lemma 6.7 states that a position q labeled a that follows p is a witnesses for color a in *some ancestor* of p. Thus, if two a-labeled positions q and q' follow a same position p (in other words: (q, q') is a witness for non-determinism), then q and q' are witness for color a in some ancestors n and n' of p. In particular, one of n or n' is a strict ancestor of the other because (P1) rules out the possibility of having $n = n'$.

There may still be quadratically many such pairs q and q'. The remaining of the section further reduces the number of pairs that should be considered when searching a witness for non-determinism. Essentially, for every node n of color a, we shall identify three positions whose combination may build a witness for non-determinism: the witness for color a in n, the unique a-labeled position in $First(n)$ if any, and some other specific position.

[2] the witness for color (a position) should not be confused with a witness for non-determinism (a pair of position)

We say that a node $n \in N_e$ has *class* a if n has color a, or n is a position labeled a, or n is the lowest common ancestor of two nodes of class a. The *a-skeleton* t_a of e consists of all nodes n of class a plus their *SupLast* and *pStar* nodes (as defined in Section 3.1.3). The node labels in t_a are taken over from e, and the tree structure is inherited from e: n' is the left (resp. right) child of n in t_a if (1) n' is in the subtree of the left (resp. right) child of n in e, (2) $n \preccurlyeq n'$, and (3) there is no n'' in t_a with $n \preccurlyeq n'' \preccurlyeq n'$. If a node has no left (resp. right) child defined in this way, then the corresponding pointer is set to *Null*. Note that a node in t_a can be labeled $\odot$ or $+$ and have its left (or right) child point to *Null*. Figure 6.2 presents a regular expression and its a-skeleton. Our skeleta are very similar to the skeleta from [BP11], and so they can all be computed in linear time:

Lemma 6.9. *The collection of a-skeleta for all $a \in \Sigma$ can be computed in time $O(|e|)$.*

Proof. The size of the a-skeleton is linear in the number of positions labeled a in e. Hence the size of the collection of a-skeleta is linear in $|e|$. The skeleta can be constructed in linear time by simply applying LCA repeatedly, inserting each position from e in left-to-right order using the linear preprocessing so that the LCA of two nodes of e is obtained in constant time. This construction is detailed in Proposition 4.4 of [BP11]. □

In the a-skeleton t_a, we equip each node n with three pointers: *Witness*(n, a), *FirstPos*(n, a), and *Next*(n, a). For every node n in t_a,

- if n has color a then *Witness*(n, a) is the witness for color a in n (and is undefined otherwise)
- *FirstPos*(n, a) is the position p labeled a such that $p \in$ *First*(n) if it exists (and is undefined otherwise); note that property (P1) guarantees that there is at most one such position p
- *Next*(n, a) is the set of all positions in *FollowAfter*$_e(n)$ labeled a.

The set *FollowAfter*$_e(n)$ is an extension of *Follow* to internal nodes n of e,

$$\mathit{FollowAfter}_e(n) = \{q \mid n \not\preccurlyeq q \text{ and } \exists p \in \mathit{Last}(n). q \in \mathit{Follow}_e(p)\}.$$

Constructing the data structures *FirstPos* and *Witness* is straightforward: *Witness* is built simultaneously with the a-skeleton; *FirstPos* can for instance be computed in a single bottom-up traversal of each a-skeleton, using pointers *SupFirst* from e and ancestor queries in e. Let n be the root node of the a-skeleton. Then *BuildNext*$(a, n, \varnothing)$ in Algorithm 2 builds the data structure *Next*(n', a) for all nodes n' of the a-skeleton.

Algorithm 2: Computing *Next*(n, a), if e is deterministic.

```
procedure BuildNext($a : \Sigma$, $n$ : Node, $Y$ : Set(Node)) : Bool
 1  if $SupLast(n)$
 2      then $Y \leftarrow \varnothing$
 3  if $n$ is the left child in $t_a$ of a $\odot$-node and
 4     $n$ has a right sibling $n'$ in $t_a$ and
 5     $(\neg SupLast(n)$ or $parent_{t_a}(n) = parent_e(n))$
 6      then $Y \leftarrow Y \cup \{FirstPos(n')\}$
 7  $Next(n, a) \leftarrow \{p \in Y \mid n \not\preccurlyeq p\}$
 8  if $lab(n) = *$
 9      then $Y \leftarrow Y \cup \{FirstPos(n, a)\}$
10  if $|Y| > 2$
11      then return false
12  if $Lchild_{t_a}(n) = Null$
13      then return true
14      else $B \leftarrow BuildNext(a, Lchild_{t_a}(n), Y)$
15  if $Rchild_{t_a}(n) = Null$
16      then return $B$
17      else return $B \wedge BuildNext(a, Rchild_{t_a}(n), Y)$
end procedure
```

Lemma 6.10. *Calling BuildNext*$(n, a, \varnothing)$ *for each* $a \in \Sigma$ *and root node* n *of* t_a *takes in total time* $O(|e|)$*. If any call returns* false *then* e *is non-deterministic. Otherwise, the set Next*(n, a) *defined during the execution consists of all positions in FollowAfter*$_e(n)$ *labeled* a*, for* $n \in N_{t_a}$ *and* $a \in \Sigma$*.*

Proof. The $O(|e|)$ time is achieved because (1) *BuildNext* is called at most m-times, where m is the number of nodes of all skeleta, and $m \in O(|e|)$ by Lemma 6.9, and (2) each line of the algorithm runs in constant time because $|Y| \leqslant 2$ at each call, due to Line 10. To see the correctness consider the execution along a path in t_a. If at Line 7 the current node n has an ancestor u labeled $*$ with no *SupLast*-node on their path, then Y contains *FirstPos*(u); if n is in the left subtree of an ancestor u labeled $\odot$ with no *SupLast*-node on their path, and n has a right sibling n' in t_a, then Y contains *FirstPos*(n'). Together with Line 7, these conditions are equivalent to *FirstPos*$(u) \in$ *FollowAfter*$_e(n)$. Clearly, e is non-deterministic if $|Y| > 2$ in Line 10. □

We define another condition:

(P2) for every $a \in \Sigma$ and $n \in N_{t_a}$, *Next*(n, a) contains at most one element.

Clearly, (P2) can be tested in linear time (for instance by incorporating it into Algorithm 1). If (P2) is false, then e is non-deterministic. Thus, from now on we assume that both (P2) and (P1) are true. We identify $\mathit{Next}(n, a)$ with q if $\mathit{Next}(n, a) = \{q\}$, and let it be undefined otherwise.

Lemma 6.11. *Let $p, q \in \mathit{Pos}(e)$ with $\mathit{lab}_e(q) = a$. If $q \in \mathit{Follow}_e(p)$ then the lowest ancestor n of p having color a exists and satisfies $q = \mathit{Witness}(n, a)$ or $q = \mathit{FirstPos}(n, a)$ or $q \in \mathit{Next}(n, a)$.*

Proof. By Lemma 6.5, Lemma 6.7 (1), and Lemma 6.10: $q = \mathit{Witness}(n, a)$ if $\mathit{Rchild}(n) \preccurlyeq_e q$, $q = \mathit{FirstPos}(n, a)$ if $\mathit{Lchild}(n) \preccurlyeq_e q$, and $q = \mathit{Next}(n, a)$ if $n \not\preccurlyeq_e q$. □

From Lemma 6.11 and the definition of (P1) and (P2) we obtain that an expression e is non-deterministic iff one of the following three conditions is satisfied: (1) (P1) is false, (2) (P2) is false, or (3) there exist $a \in \Sigma$, $n \in N_{t_a}$ of color a, and two distinct positions q, q' in $\{\mathit{FirstPos}(n, a), \mathit{Witness}(n, a), \mathit{Next}(n, a)\}$ such that $\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(q') \neq \varnothing$.

Furthermore, we prove that the case where both q and q' are different from $\mathit{Witness}(n, a)$ need not be considered. Let F and N denote the nodes $\mathit{Next}(n, a)$ and $\mathit{FirstPos}(n, a)$, and let n_F and n_N denote the parent of their *SupFirst*-node. We can prove that either $n_F \preccurlyeq n_N \preccurlyeq n$, in which case $F = \mathit{FirstPos}(n_N, a)$ (and $N = \mathit{Witness}(n_N, a)$), or $n_N \preccurlyeq n_F \preccurlyeq n$, in which case N is one of $\mathit{FirstPos}(n_F, a)$ or $\mathit{Next}(n_N, a)$ (and $F = \mathit{Witness}(n_F, a)$). We have thus proved that in an expression that satisfies (P1) and (P2), every witness for non-determinism (q, q') with $\mathit{lab}(q) = a$ consists of the witness for color a in some node n together with one of $\mathit{FirstPos}(n, a)$ or $\mathit{Next}(n, a)\}$.

Lemma 6.12. *The expression e is non-deterministic iff* (P1) *is false,* (P2) *is false, or there exist $a \in \Sigma$, a node $n \in N_{t_a}$ of color a, and a position q in $\{\mathit{FirstPos}(n, a), \mathit{Next}(n, a)\}$ such that $\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(\mathit{Witness}(n, a))$ contains at least one position.*

Algorithm Determinism Testing

To check determinism using Lemma 6.12 we need to check for $a \in \Sigma$ and $n \in N_{t_a}$ of color a, and for every position q in $\{\mathit{FirstPos}(n, a), \mathit{Next}(n, a)\}$ whether or not

$$\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(\mathit{Witness}(n, a)) \neq \varnothing.$$

Two combinations can occur for a position p:

(1) $\mathit{Witness}(n, a)$ and $\mathit{Next}(n, a)$ follow p, or

(2) $\mathit{Witness}(n, a)$ and $\mathit{FirstPos}(n, a)$ follow p, or

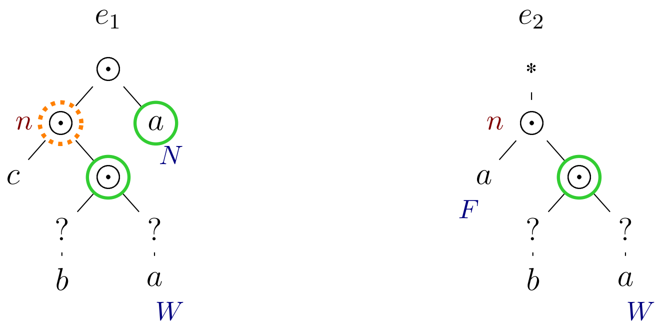


Figure 6.4.: Combinations (1) and (2).

To understand the first combination, consider the expression $e = (c(b?a?))a$, and let n be the parent of the c node in e. Thus, n is of color a, with the left a in e as witness. Clearly e is non-deterministic: take p as the c position, then both $\mathit{Witness}(n, a)$ and $\mathit{Next}(n, a)$ follow p. The same holds for the expressions $e' = (c(a?b?))a$ and $e'' = (c(b?a)^*)a$. However, expression $e''' = (c(b?a))a$ is deterministic; this is because n's right subtree is non-nullable, which prevents that $\mathit{Next}(n, a)$ and $\mathit{Witness}(n, a)$ both follow a same position p. It is not hard to see, and is formally shown in the proof of Theorem 6.13, that the first combination occurs if and only if the right-child of n is nullable.

Let us now consider combination (2). This combination can only occur if there is a $*$-node $S = \mathit{pStar}(n)$ above n, and $\mathit{SupLast}(n)$ is above this node S. Let $e = (a(b?a))^*$ and let n be the parent of the first a-position. As we can see, this expression is deterministic. This is for a similar reason as before: because the right child of n is non-nullable. If we consider $e' = (a(b?a?))^*$ then this expression is indeed non-deterministic and it holds that both $\mathit{FirstPos}(n, a)$ and $\mathit{Witness}(n, a)$ follow position p, where p is for instance the b-position. Thus, combination (2) requires that the right child of n is nullable, and also that $\mathit{FirstPos}(S, a) = \mathit{FirstPos}(n, a)$. The latter guarantees that on the path from S to $\mathit{FirstPos}(n, a)$ there is nothing non-nullable "to the left", and hence, that $\mathit{FirstPos}(n, a)$ follows the same position p that $\mathit{Witness}(n, a)$ follows.

To check determinism of e we check (P1), (P2), and then we execute for every $a \in \Sigma$ and every node n with color a, $\mathit{CheckNode}(n, a)$ of Algorithm 3; if any call returns false, then e is non-deterministic.

Theorem 6.13. *Determinism of a regular expression e can be decided in time $O(|e|)$.*

Proof. Let S, W, N, and F denote the nodes $\mathit{pStar}(n)$, $\mathit{Witness}(n, a)$, $\mathit{Next}(n, a)$, and $\mathit{FirstPos}(n, a)$ respectively. Since (P1) and (P2) can be tested in $O(|e|)$ time, it suffices, by Lemma 6.12, to prove the following two statements.

(i) $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N) \neq \varnothing$ iff $\mathit{Rchild}_e(n)$ is nullable and $N \neq \mathit{Null}$,

(ii) $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(F) \neq \varnothing$ iff $F \neq \mathit{Null}$, $S \neq \mathit{Null}$, $\mathit{Rchild}_e(n)$ is nullable, $\mathit{FirstPos}(S, a) = F$, and $\mathit{SupLast}(n) \preccurlyeq S$.

Let us prove statement (i) first. If $N \neq \mathit{Null}$ and $\mathit{Rchild}_e(n)$ is nullable then $\mathit{Lchild}_e(n)$ is not a *SupLast*-node. Therefore any position in $\mathit{Last}(\mathit{Lchild}_e(n))$ belongs to $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N)$. For the only-if direction, let q be a position in $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N)$. Then in particular $N \neq \mathit{Null}$. Node n is a strict ancestor of q since $q \in \mathit{Follow}_e^{-1}(W)$ and $n = \mathit{parent}_e(\mathit{SupFirst}(W))$. As q belongs to $\mathit{Follow}_e^{-1}(N)$, $\mathit{SupLast}(q)$ is an ancestor of n. This implies that $\mathit{Rchild}(n)$ is nullable according to Lemma 6.8, since $\mathit{Rchild}(n) = \mathit{SupFirst}(W)$ and W follows q.

We now prove (ii). If $F \neq \mathit{Null}$, $\mathit{Rchild}_e(n)$ is nullable, $\mathit{FirstPos}(S, a) = F$, and $\mathit{SupLast}(n) \preccurlyeq S$, then any q in $\mathit{Last}(\mathit{Lchild}(n))$ belongs to $(\mathit{Follow}_e^{\odot})(W) \cap (\mathit{Follow}_e^{*})^{-1}(F)$. Conversely, let q be a position in $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(F)$. As q belongs to $\mathit{Follow}_e^{-1}(W)$, node n is a strict ancestor of q. If $\mathit{Rchild}_e(n) \preccurlyeq_e q$ then $q \in (\mathit{Follow}_e^{*})^{-1}(F)$, hence $\mathit{FirstPos}(S, a) = F$ and $\mathit{SupLast}(n) \preccurlyeq S$, and furthermore $\mathit{SupLast}(q) \preccurlyeq S$, so that $\mathit{Rchild}_e(n)$ is nullable according to Lemma 6.8. Assume now that $\mathit{Lchild}_e(n)$ is an ancestor of q, and let $x = \mathit{LCA}(q, F)$. As an ancestor of both q and F, $\mathit{Lchild}_e(n)$ is an ancestor of x. Furthermore, there is no *SupLast*-node between q and $\mathit{Lchild}_e(n)$, except possibly $\mathit{Lchild}_e(n)$, and there is no *SupFirst*-node between F and $\mathit{Lchild}_e(n)$. Consequently, x is non-nullable because $\mathit{Lchild}_e(n)$ is, and, there is no $*$-labeled node between x and $\mathit{Lchild}_e(n)$. Hence $q \notin (\mathit{Follow}_e^{\odot})^{-1}(F)$, and, more generally, $\mathit{Follow}_e^{-1}(W) \cap (\mathit{Follow}_e^{\odot})^{-1}(F)$ is empty. This means that $q \in (\mathit{Follow}_e^{*})^{-1}(F)$. Thus $S = \mathit{pStar}(x)$ is not *Null*, satisfies $\mathit{FirstPos}(S, a) = F$, and is an ancestor of n since there is no $*$-labeled nodes between x and $\mathit{Lchild}_e(n)$. Accordingly, $\mathit{SupLast}(q) \preccurlyeq S$ and hence $\mathit{Rchild}_e(n)$ is non-nullable. □

Algorithm 3: Checking determinism.

```
procedure $\mathit{CheckNode}(n : \text{Node}, a : \Sigma) : \text{Bool}$
    $F \leftarrow \mathit{FirstPos}(n, a)$
    $S \leftarrow \mathit{pStar}(n)$
    if $\mathit{Rchild}_e(n)$ is nullable and
       $(\mathit{Next}(n, a) \neq \mathit{Null}$ or
        $(\mathit{FirstPos}(S, a) = F$ and $\mathit{SupLast}(n) \preccurlyeq S))$
         then return false
    return true
end procedure
```

Alternative Determinism Test

Determinism of e can be formulated as follows:

$$\neg(\exists p, p_1, p_2 \in Pos(e).\ lab_e(p_1) = lab_e(p_2) \wedge p_1 \in Follow_e(p) \wedge p_2 \in Follow_e(p)).$$

A natural question arises: Is there a logic that allows to capture determinism, and at the same time, has efficient model checking that yields a procedure for checking determinism in linear time? The answer is positive: It is possible with $\mathcal{X}^{=}_{reg}$, the language of Regular XPath expressions with data equality tests for binary trees with data values as defined in [BP11].

Trees with data values allow to store with every node its label, drawn form a finite set, and additionally, a data value, drawn form an infinite set. Regular XPath allows to navigate the nodes of the tree using regular expressions of simple steps (e.g., parent to the left child) and filter expressions. Filter expressions with data equality allow essentially to test whether two nodes have the same data value. In [BP11] Bojańczyk and Parys show that an $\mathcal{X}^{=}_{reg}$-expression φ can be evaluated over a tree t in time $2^{O(|\varphi|)} \times |t|$.

We wish to construct an $\mathcal{X}^{=}_{reg}$-expression φ_{det} that captures determinism and whose size is constant i.e., does not depend on the regular expression e. The main challenge is to handle position labels of e that can be drawn from an alphabet of arbitrary size: with *MSO* formulae as defined on page 89 and hence without data equality, determinism cannot be defined independently from the alphabet. This is accomplished by: 1) storing the labels of positions of e as data values and 2) using data equality to check whether two positions have the same label.

This provides an alternative and shorter proof for the possibility to test in linear time determinism of a regular expression. Yet expression φ_{det} uses the transitive closure operator of Regular XPath. It therefore does not belong to the basic fragment of XPath that Bojańczyk and Parys [BP11] can evaluated with complexity $O(|\phi|^3 \times |t|)$. We believe that our algorithm is easier to implement than the $2^{O(|\phi|)} \times |t|$ time algorithm, and that it runs more efficiently in practice. The latter claim has not been verified yet because no implementation of [BP11] is available.

Theorem 6.14. *There exists an $\mathcal{X}^{=}_{reg}$-expression φ_{det} such that for any alphabet Σ and any regular expression e over Σ, φ_{det} is satisfied in e if and only if e is deterministic.*

Proof. We present only the construction of φ_{det}. Let *SupFirst* and *SupLast* denote $\mathcal{X}^{=}_{reg}$-expressions that are satisfied only in *SupFirst*- and *SupLast*-nodes, respectively. We also use axis from-left with the same semantics as in [BP11], i.e., it goes from a node to its parent and checks that this parent has two children, of which the original node is the leftmost. Similarly, to-right goes from a node to its right child provided the original node has two children.

$$
\begin{array}{ll}
D = (\Downarrow/[\,\mathsf{not}\,\mathit{SupFirst}])^*/P & P = [\,\mathsf{not}\Downarrow] \\
U = ([\,\mathsf{not}\,\mathit{SupLast}]/\Uparrow)^* & F = ([\mathrm{lab}() = \odot])/\text{to-right}/D
\end{array}
$$

$$
\begin{aligned}
\varphi_{\odot\odot} &= \Downarrow^*/[\,\mathsf{not}\mathit{SupLast}]/\text{from-left}/[F = (\,U/\text{from-left}/F)] \\
\varphi_{**} &= \Downarrow^*/[\mathrm{lab}() = *]/[D = (U/[\mathit{SupFirst}]/\Uparrow/U/[\mathrm{lab}() = *]/D)] \\
\varphi_{\odot *} &= \Downarrow^*/[\,\mathsf{not}\,\mathit{SupLast}]/\text{from-left}/[(\text{to-right}/[\mathit{SupFirst}]/D) = (\Uparrow/U/[\mathrm{lab}() = *]/D)] \\
&\qquad \cup \Downarrow^*/[\mathrm{lab}() = *]/[D = (U/\text{from-left}/F)] \\
\varphi_{P_1} &= \Downarrow^*/[(\text{to-left}/[\,\mathsf{not}\,\mathit{SupFirst}]/D) = (\text{to-right}/[\,\mathsf{not}\,\mathit{SupFirst}]/D)] \\
\varphi_{det} &= [\,\mathsf{not}(\varphi_{P_1}\ \mathsf{or}\ \varphi_{\odot\odot}\ \mathsf{or}\ \varphi_{\odot *}\ \mathsf{or}\ \varphi_{*\odot}\ \mathsf{or}\ \varphi_{**})].
\end{aligned}
$$

Basically, φ_{P_1} checks if (P1) is violated in e and the expression $\varphi_{\ell\ell'}$ for $\{\ell, \ell'\} \subseteq \{*, \odot\}$ checks whether there exist two distinct positions p_1 and p_2 of e such that $\mathit{lab}(p_1) = \mathit{lab}(p_2)$ and $(\mathit{Follow}_e^{\ell})^{-1}(p_1) \cap (\mathit{Follow}_e^{\ell'})^{-1}(p_2) \neq \varnothing$.

Technically, the five formula have the form $\varphi = \psi/[\psi_1 = \psi_2]$ for some $\mathcal{X}\mathit{Reg}$ expressions ψ, ψ_1 and ψ_2. For any tree e and $n \in N_e$, $(\mathit{root}_e, n) \in [\![\varphi]\!]_e$ if and only if there exist $p_1, p_2 \in \mathit{Pos}(e)$ such that the following four conditions are satisfied: (1) $(\mathit{root}_e, n) \in [\![\psi]\!]_e$ (2) $(n, p_1) \in [\![\psi_1]\!]_e$ (3) $(n, p_2) \in [\![\psi_2]\!]_e$ (4) p_1 and p_2 have the same label. Let e an expression that satisfies P_1 and consider $\varphi_{\odot\odot} = \psi/[\psi_1 = \psi_2]$. If there exist a position p and two distinct positions p_1 and p_2 of e such that $\mathit{lab}(p_1) = \mathit{lab}(p_2)$ and $p \in (\mathit{Follow}_e^{\ell})^{-1}(p_1) \cap (\mathit{Follow}_e^{\ell'})^{-1}(p_2)$, then one of $n_1 = \mathit{LCA}(p, p_1)$ or $n_2 = \mathit{LCA}(p, p_2)$ is a strict descendant of the other. Assume for instance that it is n_1 that is a descendant of $\mathit{Lchild}(n_2)$. Clearly, p belongs to $\mathit{Last}(\mathit{Lchild}_e(n_1))$, so is a descendant of $\mathit{Lchild}_e(n_1)$. It also belongs to $\mathit{Last}(\mathit{Lchild}_e(n_2))$, so that $\mathit{Lchild}_e(n_1)$ is not a $\mathit{SupLast}$ node, hence $(\mathit{root}_e, n_1) \in [\![\psi]\!]_e$. Furthermore, p_1 and p_2 belong to the First-set of $\mathit{Rchild}_e(n_1)$ and $\mathit{Rchild}_e(n_2)$, respectively. Consequently, $(n_1, p_1) \in [\![\psi_1]\!]_e$ and $(n_1, p_2) \in [\![\psi_2]\!]_e$, hence $\varphi_{\odot\odot}$ is satisfied. Conversely, assume that $\varphi_{\odot\odot}$ is satisfied by e, and let n, p_1 and p_2 be nodes of e satisfying the conditions (1) to (4) above. We show easily that for any position p in $\mathit{Last}(\mathit{Lchild}(n))$, p_1 and p_2 both belong to $\mathit{Follow}_e^{\odot}(p)$. □

Testing Numeric Occurrences

Regular expression occurring in XMLSchema may contain numeric occurrence indicators. Kilpelaïnen and Tukhanen [KT07] provide an astute characterization of deterministic expressions with numeric occurrences. They deduce a polynomial algorithm to check determinism of such expressions, essentially computes a relation based on *Follow* taking numeric occurrences into account. This algorithm has cubic complexity $O(|\Sigma| \times |e|^2)$ when the size of the alphabet is not bounded. Kilpelaïnen [Kil11] improves the complexity to $O(|e| \times |\Sigma|)$. The algorithm from Kilpelaïnen is therefore quadratic ($O(|e|^2)$)

when the alphabet is not bounded. [3] Kilpelaïnen obtains this complexity by a merging-based examination of *First* and *Follow* sets, similar to the approach in [KT07], but relying on a more careful analysis of the *Follow* sets. After Theorem 3.3 in [Kil11], the author leaves as an open question whether a better complexity can be obtained, and observes that with merging-based approaches it seems difficult to go below $O(n \times |\Sigma|)$. Our algorithm essentially avoids the computation and merging of *First* and *Follow* sets, which allows us to obtain linear complexity to test determinism in the absence of numeric occurrences, for arbitrary large alphabets. We show in the appendix that our algorithm can be combined with the characterization from [KT07] to obtain a linear algorithm testing the determinism of regular expressions with numeric occurrences.

Theorem 6.15. *Determinim of a regular expression e with numeric occurrences can be tested in linear time $O(|e|)$, for an arbitrary alphabet.*

To conclude these remarks on deterministic regular expressions with numeric occurrences, let us observe that deterministic regular expressions with counters are strictly more expressive than deterministic expressions. The definition of deterministic expressions with numeric occurrences that we consider; the one used as well in [KT07, Kil11] and the XML Schema, is sometimes called *weak* determinism. A more restrictive notion of determinism, *strong* determinism has also been investigated for regular expressions with counters, and the strongly deterministic regular expressions have the same expressivity as deterministic regular expressions (without counters) [GGM12].

6.2.2. “Determinizing” Non-deterministic Expressions

When a DTD fails the determinism check, we may wish to repair it into a schema satisfying the determinism constraint. More generally, given a regular language, we may want to compute a deterministic regular expression for this language. Unfortunately, there exist regular languages that cannot be represented with a deterministic regular expression [BKW98]: $(a+b)^*a(a+b)$, for instance. This suggests the following approach: test if there exists a deterministic representation of the language, and, if so, compute it. If there is none, the database administrator can be notified so that he modifies the schema. Brüggeman-Klein and Wood [BKW98] provide a polynomial algorithm that tests if the language of a DFA can be represented with a deterministic regular expression. We briefly review this algorithm (following the presentation from Brüggeman-Klein and Wood) because we will use this algorithm to discuss approximations.

[3] Actually, in Theorem 3.3 from [Kil11], the complexity is stated as $n^2/(\log(n))$, with n representing the size of the binary representation of the regular expression. But with our notations, this translates into a quadratic $O(|e|^2)$.

The Brüggeman-Klein and Wood algorithm (BKW) for testing determinism of a regular language assumes the input DFA $\mathcal{A}_0$ to be minimal. This assumption does not affect the complexity of the algorithm as DFA $\mathcal{A}_0$ can be minimized in time $O(|\Sigma| \times |Q| \log |Q|)$ [HU79] and, from a minimal DFA $\mathcal{A}_0$, every DFA appearing in the recursive calls of algorithm $BKW(\mathcal{A}_0)$ will be minimal. Given any automaton $\mathcal{A} = (Q, \Sigma, \Delta, \{q_0\}, F)$ and state q of $\mathcal{A}$, the *orbit* of q is the strongly connected component of $\mathcal{A}$ containing q, i.e., the state q plus every state q' such that q and q' can be reached from one another. A state q is a *gate* of its orbit if q is final, or if there is a transition from q leading to a state outside the orbit of q. $\mathcal{A}$ has the orbit property if every two gates q_1, q_2 in the same orbit satisfy the following two conditions: (1) q_1 is final iff q_2 is, and (2) for every $a \in \Sigma$ and every state q outside the orbit of q_1 and q_2, $\mathcal{A}$ has transition (q_1, a, q) iff it has transition (q_2, a, q).

When $\mathcal{A}$ is a DFA, a letter $a \in \Sigma$ is $\mathcal{A}$-consistent if there exist a state $q \in Q$ such that every final state of $\mathcal{A}$ as a transition to q labeled a. Given a set S of $\mathcal{A}$-consistent letters, the S cut of $\mathcal{A}$ is the automaton obtained from $\mathcal{A}$ by removing for every final state of $\mathcal{A}$ all its outgoing transition with label in S.

We finally define $\mathcal{A}_q$, the orbit automaton of q, for every state $q \in \mathcal{A}$. $\mathcal{A}_q$ is obtained from $\mathcal{A}$ by setting the initial state to q and restricting the states to the orbit of q. The final states of $\mathcal{A}_q$ are the gates of this orbit.

Brüggeman-Klein and Wood state that their algorithm runs with complexity quadratic in the size of the input DFA. It seems they assume a constant size alphabet, because they claim quadratic complexity for the Hopcroft minimization algorithm, and further in their paper they also claim that language $\Sigma^* w$ admits a DFA of size linear in w. Nevertheless, their proof for the quadratic complexity of algorithm BKW still holds without assumption on the size of the alphabet. The crude estimation $O(|\Sigma| \times |Q| \log |Q|)$ for the minimization algorithm is not accurate enough in this case, but the complexity of minimization was refined by Valmari and Lehtinen. They prove that a DFA $(\Sigma, Q, i, F, \Delta)$ with partial transition function Δ can be minimized in time $O(|\Delta| \log |Q|)$, using space $O(|\Delta| + |Q| + |\Sigma|)$ [VL08]. The estimation of the complexity works as follows ([BKW98]): first, the automaton is minimized, once and for all. Then for each call of BKW, the set of consistent letters can be computed in $O(|\Delta|)$. Furthermore the states of the automaton can be partitioned into disjoint orbits, in linear time $O(|\Delta|)$ using the algorithm by Tarjan to compute the strongly connected components of a graph [Tar72]. This yields the overall quadratic complexity $O(|Q| \times |\Delta|)$ for algorithm BKW.

Bex et al. proved that testing determinism of a regular language is Pspace-hard when the input is a regular expression [BGMN09] instead of a DFA. However, their proof goes through a relatively complex and long reduction from Corridor Tiling. We provide a much simpler proof in the appendix. They also leave as an open question whether the determinism of a regular language can be tested in Pspace. They argue that an approach "guess-

Algorithm 4: Algorithm $BKW(\mathcal{A})$ from [BKW98], testing if $L(\mathcal{A})$ is deterministic

```
Input: minimal DFA $\mathcal{A}$
Output: true if $\mathcal{A}$ is deterministic, false otherwise
$S \leftarrow$ The set of all $\mathcal{A}$ consistent letters
if $\mathcal{A}$ has a single state without outgoing transitions then return true
else if $\mathcal{A}$ has a single orbit and $S = \emptyset$ then return false
if $\mathcal{A}_S$ has the orbit property then
    foreach orbit $K$ of $\mathcal{A}_S$ do
        choose $q$ in $K$
        if $BKW((\mathcal{A}_S)_q)$ = false then return false
    end
else
    return false
end
```

ing" an expression before testing equivalence would not work, but do not investigate whether the algorithm from Brüggeman-Klein and Wood can be simulated in polynomial space by constructing the DFA on-the-fly. Beyond this negative result, they also observe the problem to be fixed parameter tractable in k for k-occurrence regular expressions; they observe that one can test if the language of a k-ORE can be expressed with a deterministic regular expression with complexity $O(2^{2k} \times |\Sigma|^3)$ using algorithm BKW.[4] This justifies the tractability of the algorithm deciding if there exists a deterministic expression equivalent to the input regular expression, for real-life schemata.

When there exists such a deterministic expression, Brüggeman-Klein and Wood [BKW98] provide an algorithm that computes the deterministic expression in optimal exponential time. Their algorithm takes as input a DFA and computes in time $2^{O(|Q|\log(|\Sigma|))}$ an equivalent deterministic regular expression. They prove that the smallest deterministic regular expression (if any) equivalent to a DFA may require exponential size, while the conversion from regular expressions to deterministic regular expressions (when possible) also requires an exponential blowup. Bex et al. study additional algorithms to compute a deterministic regular expression from a determinizable regular expression [BGMN09].

[4] the numbers in [BGMN09] are slightly different.

6.3. Approximation

The preceding sections show that the limited expressivity of DTDs makes it hard and sometimes unfeasible to construct a view schemata: depending on the restrictions, one has to tackle non-local or even non-regular (context-free) features in the view schema. As a way to elude these obstacles we propose to relinquish exact view schemata and resort to approximations of the view schema instead. In our opinion the primary purpose of the view schema is to guide the user in her attempt to formulate a meaningful query. In that perspective, we will consider that a good approximation of the view schema allows the user to check if the result of a query is empty for every document: that way, the user will never formulate queries which return no answer for all documents. Of course, the existence of such an approximation depends on the expressivity of the query language. We therefore propose three simple approximations which present different information on the view schema. The following paragraph surveys the size of the resulting approximations as well as the complexity to compute the approximations. We then investigate which approximation may be considered suitable when the query language ranges over several XPath fragments.

6.3.1. Subset, Subword, and Parikh Approximations

We define local approximations, that replace each production rule of a CDTD with a regular expression. Our approximations are defined as a function mapping context-free languages into (deterministic) regular expressions. This function is extended to CDTD as follows: approximation Approx() replaces a CDTD (Σ, r, P) with the DTD (Σ, r, P'), where for each $a \in \Sigma$, $P'(a) = \text{Approx}(P(a))$.

Subset Approximation As a first approximation, we only provide the user with the set of elements that can appear below a node.

Definition 6.1. *For every context-free grammar G we define the subset approximation of G as $Approx(G) = (alph(G))^*$, where $alph(G)$ denotes the set of all letters appearing in $L(G)$. Naturally, $Approx(G)$ is a deterministic expression, and its size is at most linear in $|G|$.*

Parikh Approximation The subset approximation does not even allow to derive the information of which symbols can occur simultaneously below a node. Therefore we propose a more precise approximation based on the Parikh image of the word. We recall the definition of the Parikh image, with the result from Parikh [Par66] that leads to the construction of the approximation.

Definition 6.2. *Let $\Sigma = \{a_1, \ldots, a_n\}$ an alphabet, and w a word over alphabet Σ. The Parikh image of w is the n-uple $\Phi_{Parikh}(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$. The definition is extended to a language L over Σ as follows: $\Phi_{Parikh}(L) = \{\Phi_{Parikh}(w) \mid w \in L\}$.*

Definition 6.3. *A subset of $\mathbb{N}^k$ is* linear *if it is of the form $S(\mathbf{v}, \{\mathbf{u}_1, \ldots, \mathbf{u}_n\}) = \{\mathbf{v} + k_1\mathbf{u}_1 + k_2\mathbf{u}_2 + \ldots k_n\mathbf{u}_n \mid k_1, \ldots, k_n \in \mathbb{N}\}$ for some $n \in \mathbb{N}$, and $\mathbf{v}, \mathbf{u}_1, \ldots, \mathbf{u}_n$ in $\mathbb{N}^k$. A subset of $\mathbb{N}^k$ is a* semilinear set *if it is the union of a finite number of linear sets.*

Theorem 6.16 (Parikh [Par66]). *The Parikh image of a context-free language L is a semilinear set. Therefore, one can compute a regular language with the same Parikh image as L.*

Several alternative construction have been proposed to build regular expressions or NFAs with Parikh image $\Phi_{\text{Parikh}}(L(G))$. Esparza et al. [EGKL11] survey those constructions, with the complexity expressed in terms of n and m, respectively the number of variables of the grammar $G = (V, T, S, P)$ and the *degree* of G, defined as the maximal number of variable occurrences appearing in some right-hand side of a production rule minus one: $n = |V|$ and $m = -1 + \max\{k \mid \exists A \in V, B_1, \ldots, B_k \in V, w_0, \ldots w_k \in T^*. w_0B_1w_1B_2 \ldots B_kw_k \in P(A)\}$.

Remark 6.2. *It is obvious that every CFG can be transformed in linear time into a grammar of which the production rules contain at most two characters (be they variables or terminals). Therefore, we define 2NF grammars as the context-free grammars satisfying this constraint. For our purpose in this dissertation, namely a bound in terms of $|G|$, it is sufficient to consider 2NF grammars. A fortiori these grammars will have degree $m \leqslant 1$.*

Theorem 6.17 ([EGKL11]). *Given a grammar $G = (V, T, S, P)$ of degree m with n variables, one can compute an automaton M_G with $\binom{n+nm+1}{n}$ states such that M_G and G have the same Parikh image. The alphabet of M_G is $T^{\leqslant k}$ with k the maximal number of non terminals appearing in a production rule of P.*

If the degree of G is $m = 1$, then M_G has $\binom{2n+1}{n}$ states. As observed by Esparza et al., $\binom{2n+1}{n}$ can be bounded by $O(4^n)$, therefore by $O(4^{|G|})$. For *2NF* grammars we obtain directly an automaton with $\binom{2n+1}{n} = O(4^{|G|})$ states over alphabet T by the same remark[5]. DTD D_3 of Example 6.1 provides a corresponding $\Omega(2^n)$ lower bound: any automaton accepting the language $\{a^{2^n}\}$ needs at least $2^n + 1$ states as also observed in [EGKL11].

[5] see the discussion after Theorem 3.1 in [EGKL11]

Remark 6.3. *Esparza et al. only considers the size of the resulting automaton in terms of states, there is no mention of the complexity for computing the actual automaton. However, it is clear that it can be computed in time $O(|G| + |M_G|)$. One could also avoid to build the whole automaton: after a preprocessing in $O(|G|)$ one can decide in time $O(|V|)$ if there is a transition from q to q' labeled "a" for any pair of states q, q' in M_G and any $a \in \Sigma$.*

Esparza et al. also implicitly provide an upper bound toward regular expressions. More accurately, they observe that their construction can be plugged into a recent result by To(Lin)[6] in order to bound the size of the semilinear representation of G.

Theorem 6.18 ([To10a]). *Let $\mathcal{A}$ an NFA with s states over an alphabet Σ of size k. Then, there exists a representation of $\Phi_{Parikh}(\mathcal{A})$ as a union of $O(s^{k^2+3k+3}k^{4k+6})$ linear sets, with each linear set $S(\mathbf{v}, \{\mathbf{u}_1 \ldots \mathbf{u}_j\})$ satisfying the following three properties: (1) $j \leqslant k$, (2) each $\mathbf{u}_i$ belongs to $\{0, \ldots s\}^k$ and (3) the maximal entry in $\mathbf{v}$ is bounded by $s^{3k+3}k^{4k+6}$. Furthermore, this semilinear set can be computed from $\mathcal{A}$ in time $2^{O(k^2 \log(ks))}$.*

Esparza et al.[5] discuss how the value of s can be obtained from M_G by first introducing intermediate states to obtain alphabet T instead of $T^{\leqslant k}$. For *2NF* grammars, we get $s \leqslant |G|\binom{2n+1}{n} \leqslant O(|G| * 4^{|G|})$.

Corollary 6.19 (from [EGKL11]). *Let G a 2NF CFG with n variables over alphabet Σ of size k. Set $s = |G|\binom{2n+1}{n}$. Then, there exists a representation of $\Phi_{Parikh}(L(G))$ as a union of $O(s^{k^2+3k+3}k^{4k+6})$ linear sets, with each linear set $S(\mathbf{v}, \{\mathbf{u}_1 \ldots \mathbf{u}_j\})$ satisfying the following three properties: (1) $j \leqslant k$, (2) each $\mathbf{u}_i$ belongs to $\{0, \ldots s\}^k$ and (3) the maximal entry in $\mathbf{v}$ is bounded by $s^{3k+3}k^{4k+6}$. Furthermore, this semilinear set can be computed from G in time $2^{O(k^2 \log(ks))}$.*

For every semilinear set S, a regular expression e satisfying $\Phi_{\text{Parikh}}(L(e)) = S$ can be trivially computed in linear time. Therefore we obtain the same bounds as above for regular expressions instead of semilinear sets. Using the approximation of $\binom{2n+1}{n}$ by $O(4^{|G|})$, we obtain an upper bound of $2^{O(k^2 \log(k \times |G| \times O(4^G)))} = 2^{O(k^2 \log(k)) + O(k^2 \times |G|)} = 2^{O(k^2 \times |G|)}$.

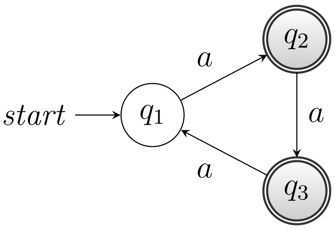
Corollary 6.20. *Let G a CFG over alphabet Σ. One can compute a regular expression $\mathscr{P}(G)$ with $\Phi_{Parikh}(L(\mathscr{P}(G))) = \Phi_{Parikh}(L(G))$ in time $2^{O(|\Sigma|^2 \times |G|)}$.*

Definition 6.4. *Given a context-free grammar G, we define the Parikh approximation of G as $Approx(G) = \mathscr{P}(G)$.*

[6] A version of this result also appears in [To10b], but the constants in[To10a] are slightly better, and that result does not appear in the paper [KT10] published after merging [To10b] with related results by Kopczyński. We also observe that the bound is only a rough estimation: a simple analysis of the proof shows that the degree of the polynomials can be slightly lowered.

We may want to approximate the schema with an XML DTD. Then, the above corollary is not fully satisfying as we are looking for a deterministic regular expression. Unfortunately, deterministic regular expressions are strictly less expressive than regular expressions with respect to Parikh image: for languages over unary alphabet, the Parikh image of the language is essentially the language itself, in the sense that two different words have different image. The following example shows that even over a unary alphabet, deterministic regular expressions cannot represent all regular languages, even though deterministic automata over unary alphabet always consist of one loop preceded by a single tail, where the loop or the tail may possibly be empty.

Example 6.4. *Set $\Sigma = \{a\}$ the alphabet. The DFA $\mathcal{A}_3$ on the right is clearly minimal. However, Algorithm 4 proves there is no deterministic regular expression accepting $L(\mathcal{A}_3)$, because $\mathcal{A}_3$ consists of a single non-trivial loop, and letter a is not $\mathcal{A}_3$-consistent.*



Automaton $\mathcal{A}_3$

Subword Approximation Parikh approximation records the number of occurrences of each symbol, however it may fully reorder the elements. This does not fit when the order of the siblings is relevant. We therefore introduce a last approximation, that does not preserve the Parikh image of the productions, but preserves the sibling ordering.

This last approximation uses subwords and relies on a result from Courcelle [Cou91]. Formally, u is a *subword* of w, denoted $u \sqsubseteq w$, if $u = u_1 \cdots u_k$ and there exist $v_0, v_1, \ldots, v_k \in \Sigma^*$ such that $v_0 u_1 v_1 \cdots v_{k-1} u_k v_k = w$. Courcelle shows that for every CFG G one can construct a regular expression $G_\downarrow$ accepting the subwords of $L(G)$:

Theorem 6.21 ([Cou91]). *For every CFG G one can construct a regular expression $G_\downarrow$ such that $L(G_\downarrow) = \{u \mid \exists w \in L(G). u \sqsubseteq w\}$.*

Remark 6.4. *The construction given by Courcelle is effective and runs in exponential time $2^{O(|G|)}$.* [7]

[7] There is quite an interesting story about these Theorem 6.21 and Remark 6.4: regularity of the subword closure is actually a simple consequence of Higman's Lemma [Hig52], and so can also be seen as a consequence of the Robertson-Seymour theorem. The paper by Courcelle proves that a regular expression representation can be effectively obtained for CFG. Apparently unknown to Courcelle, van Leeuwen had already proved the effectiveness of the construction of an NFA for the subword closure of a language. However, his algorithm is more complex than the one from Courcelle. Courcelle does not analyse the complexity of his algorithm but it is relatively easy to bound the running time by $2^{O(|G|)}$. Atig et al. [ABQ09] already mention that Courcelle's algorithm

Proof. We refer the reader to the construction in [Cou91]. To establish the complexity of his algorithm, we observe that his algorithm provides a straight line program of size $O(|G|)$ over the alphabet of regular expressions: $\Sigma \cup \{+, (,), {}^*\}$. The word represented by this straight line program is a regular expression with language $\{u \mid \exists w \in L(G). u \sqsubseteq w\}$. The size of this expression is therefore $2^{O(|G|)}$. □

Remark 6.5. *This construction is optimal since for every $n \in \mathbb{N}$ there exists a grammar G of size $O(n)$ accepting a^{2^n}, and every NFA accepting $\{u \mid \exists w \in L(G). u \sqsubseteq w\}$ has size at least 2^n. As regular expressions over a fixed alphabet can be converted to NFAs in linear time, this implies the same lower bound toward regular expressions instead of NFAs.*

The regular expression $G_\downarrow$, however, is not deterministic in general. Nevertheless, every subword-closed language L_0 can be represented by a deterministic regular expression. Indeed, the minimal DFA accepting a language L_0 such that $L_0 = \{u \mid \exists w \in L_0. u \sqsubseteq w\}$ has only trivial loops (or no loops at all). Therefore, L_0 can be represented with a deterministic expression according to the algorithm from Brüggeman-Klein and Wood (see [BKW98]).

Lemma 6.22. *Given any NFA $\mathcal{A}$ with n states, we can build in time $2^{O(n \log |\Sigma|)}$ a deterministic regular expression e that accepts the subwords of $L(\mathcal{A})$.*

Proof. Let $\mathcal{A} = (\Sigma, Q, \{q_0\}, F, \Delta)$ an NFA with $|Q| = N$ states. Set $\mathcal{A}_* = (\Sigma, Q, \{q_0\}, Q, \Delta_*)$ the NFA obtained from $\mathcal{A}$ by making every state final and replacing Δ with Δ_* defined as follows. For every q, q', $(q, a, q') \in \Delta_*$ iff there are $n \geqslant 1$, $q_0 = q, q_1, \ldots, q_n = q' \in Q$ and $a_1, \ldots, a_n \in \Sigma$ such that (1) for every $i \leqslant n$ $(q_{i-1}, a_i, q_i) \in \Delta$ and (2) there exists $j \leqslant n$ such that $a_j = a$. Intuitively, Δ_* can be obtained by adding an ϵ-transition in parallel to every transition from Δ, and then removing all ϵ-transitions with the usual transitive closure algorithm. This is a fairly standard construction, see also [GHK09] for instance.

We observe that the strongly connected components of $\mathcal{A}_*$ are cliques, and for every such clique K, there exists a set of letters $\mathcal{S}$ such that for every q, q' in K, (1) there is no transition with label $b \notin \mathcal{S}$ from q to q', (2) for every $a \in \mathcal{S}$, there is a transition with label a from q to q', and (3) q and q' share the same outgoing and incoming transitions with states outside their clique K. Therefore, we merge in $\mathcal{A}_*$ all the states q, q' in the same strongly connected component into a single state. Consequently Δ_* induces a partial

has complexity "exponential" in $|G|$. Gruber, Holzer and Kutrib undertook a thorough investigation of the subword closure of word languages([GHK07, GHK09]) and they evaluate the complexity of the construction from Van Leeuwen to $O(n2^{\sqrt{2^n}\log n})$ in [GHK09]. The result from Courcelle closes the gap between a $2^{\Omega(n)}$ lower bound and the $O(n2^{\sqrt{2^n}\log n})$ upper bound in [GHK09], modulo the constant hidden in the Landau notation.

order $\preccurlyeq$ on Q, defined by: $q' \preccurlyeq q$ iff $q = q'$ or there exists some $a \in \Sigma$ such that $(q, a, q') \in \Delta_*$.

We build the DFA obtained from $\mathcal{A}_*$ by the powerset construction; the states of $\mathcal{A}_*$ are subsets of Q, and all the strongly connected components of $\mathcal{A}_*$ consist of a single state. The automaton $\mathcal{B}$ is obtained from this powerset automaton by identifying each state (a subset of Q) with its set of maximal elements for $\preccurlyeq$. The point is that we are going to unfold this automaton into an equivalent DFA whose underlying graph is a tree (plus trivial loops on some states), but we need an argument to bound the depth of this tree. This is why we identify each state of $\mathcal{B}$ with an antichain in Q for $\preccurlyeq$. Let (S, a, S') a transition from $\mathcal{B}$. Then for every $q \in S, q' \in S'$ we have either $q' \preccurlyeq q$ or q and q' are incomparable. Furthermore, for every $q' \in S'$ there exists some $q \in S$ such that $q' \preccurlyeq q$. Consequently, every run of $\mathcal{B}$ goes through at most n different states: whenever the state S changes, there exists some $q \in Q$ which is removed from S and which satisfies $q \not\preccurlyeq q'$ for every $q' \in S$, therefore q' cannot come back in the future states. The tree (with trivial loops) obtained by unfolding $\mathcal{B}$ has rank at most $|\Sigma|$ and depth at most n, and therefore allows to derive inductively a deterministic regular expression in time $2^{O(n \log |\Sigma|)}$. □

Proposition 6.23. *For every CFG G one can construct a deterministic regular expression $G_\downarrow^{det}$ such that $L(G_\downarrow^{det}) = \{u \mid \exists w \in L(G). u \sqsubseteq w\}$. Furthermore, $G_\downarrow^{det}$ can be computed in time $2^{2^{O(|G|)}}$.*

Proof. From Theorem 6.21 we immediately get a doubly exponential upper bound toward DFAs, but the translation from DFAs to deterministic regular expressions involves yet another exponential. The following sketch of algorithm computes $G_\downarrow^{det}$ in doubly exponential time. We first compute the Glushkov automaton for the regular expression $G_\downarrow$ from Courcelle's algorithm in Theorem 6.21. This automaton has $2^{O(|G|)}$ states. Then Lemma 6.22 allows to compute a deterministic expression equivalent to this NFA in time $2^{2^{O(|G|)} \times \log |\Sigma|} = 2^{2^{O(|G|)}}$. □

This upper bound can be matched with a doubly exponential lower bound: we can build subword-closed context-free languages for which the translation into a DFA requires two exponentials, hence a doubly exponential lower bound for deterministic regular expressions. This rules out the use of subword closure on arbitrary grammars, but subword closure may still lead to reasonable approximations on practical cases.

Proposition 6.24. *For every n, we can build a CFG G of size $O(n)$ such that every DFA for $G_\downarrow^{det}$ needs 2^{2^n} states.*

Proof. For every natural N, we consider the following language L_N, were w^R represents the reversal of word w: $w^R = w[k]w[k-1]\dots w[2]w[1]$ for a word w of length k.

$$L_N = \{(a+b)^j w(a+b)^{2^N} \# w^R (a+b)^{2^N - j} | j \leqslant 2^N, w \in \{a,b\}^N\}$$

The following two claims establish the doubly exponential blowup.

Claim. *The subwords of L_N can be represented with a grammar G_N of size $O(N)$.*

Let G_N be the grammar (V, T, S_N, P) with terminals $T = \{a, b\}$, non-terminals $V = \{S_0, \dots, S_N, W_0, \dots, W_N, U_0, \dots, U_N, A, B\}$, and with production rules defined below for $k, i \in \{0, \dots, N-1\}$. Clearly, $|G_N| = O(N)$ and G_N accepts exactly the subwords of L_N, which concludes the proof of the first claim.

$$\begin{array}{llll} S_{i+1} \to S_i U_i \mid U_i S_i & W_{k+1} \to B W_k B \mid A W_k A & U_{i+1} \to U_i U_i & A \to a \mid \epsilon \\ S_0 \to W_N U_0 \mid U_0 W_N & W_0 \to U_N \# & U_0 \to A \mid B & B \to b \mid \epsilon \end{array}$$

Claim. *Any DFA accepting the subwords of L_N needs 2^{2^N} states.*

We prove easily the claim with the standard residual technique. Let u and u' two distinct words in $\{a,b\}^{2^N}$: u and u' differ on the i^{th} letter for some $i \leqslant 2^N$. Then u and u' lead to different states in any DFA for L_N: we exhibit a word v such that $uv \in L_N$, but $u'v \notin L_N$. Let w be the word defined by $w = u[i]u[i+1]\dots u[i+N-1]$, with the convention $u[j] = a$ for all $j > 2^N$. For $v = a^{N+i-1} \# w^R b^{2^N-(i-1)}$, we obtain $uv \in L_N$, but $uv' \notin L_N$. There are 2^{2^N} words in $\{a,b\}^{2^N}$. Consequently, any DFA accepting L_N needs at least 2^{2^N} states. This concludes the proof of the claim, and thereby the proof of Proposition 6.24. □

Definition 6.5. *Given a context-free grammar G, we define the subword approximation of G as Approx$(G) = G_{\downarrow}^{det}$.*

We assume w.l.o.g. that $G_{\downarrow}^{det}$ is uniquely determined by G. This can be obtained if we fix a deterministic algorithm for computing $G_{\downarrow}^{det}$.

Approximating Regular Expressions or NFAs To complete the picture, let us discuss the complexity of those approximations when the input language is a regular expression or a word automaton instead of a CFG. In order to represent the subwords of a regular expression or automaton with a DFA we can use the classical powerset construction, as in the first part of the proof from Lemma 6.22. The resulting DFA has $2^{O(n)}$ states where n is the number of states of the input automaton (the Glushkov automaton if the input is an expression). If we wish to represent the subwords of a regular expression

e with a deterministic regular expression, then we can apply Lemma 6.22 to its Glushkov automaton. The following Lemma shows that there cannot be a polynomial algorithm for the task. More exactly, we prove a rough superpolynomial lower bound for the operation that represents the subwords of a regular expression with a DFA. This implies in particular the same bound toward deterministic expressions instead of DFAs.

Lemma 6.25. *There exists a constant $\alpha > 0$ and a family of regular expressions e_n of size $O(n)$ such that any DFA accepting the subwords of e_n has at least 2^{n^α} states.*

Proof. This bound can be obtained from the combination of Proposition 6.24 and Theorem 6.21. Proposition 6.24 essentially states that for some constant $c > 0$, there exists for every n a CFG G_n of size at most cn such that any DFA accepting the subwords of $L(G_n)$ has at least 2^{2^n} states. Theorem 6.21 states that there is some constant $d > 0$, such that for every CFG of size cn one can build a regular expression of size at most 2^{dn}. Let us consider a transformation that maps every regular expression e to a DFA accepting the subwords of e. For every n, we denote by $f(n)$ the maximal number of states of the resulting DFA, when e ranges over all expressions of size at most n. From what precedes, we deduce that $f(2^{d(cn)}) \geqslant 2^{2^n}$, hence $f(n) \geqslant 2^{n^{1/(cd)}}$.□

Okhotin [Okh10] proves that there exists a DFA A_n of arbitrary large size n such that any DFA accepting the subwords of A_n needs $2^{n/2-2}$ states. This implies an exponential blowup for the representations by deterministic expressions or DFAs of both NFAs and DFAs. These lower bounds are matched by exponential upper bounds derived from Lemma 6.22.

When the approximation can be an arbitrary regular expression instead of a deterministic one, the approximation is simplified from regular expressions, but remains expensive from both DFAs and NFAs. From a regular expression e, we can trivially compute a regular expression for the subwords of $L(e)$ with the addition of a question mark after each letter of e. As discussed above, Lemma 6.22 allows to compute a (deterministic) regular expression for the subwords of a DFA or NFA, in exponential time. But Ellul et al. [EKSW05] establish that a classical algorithm can provide a better bound when the output can be an arbitrary regular expression.

Lemma 6.26 (Corollary 18 in [EKSW05]). *If A is an NFA with n states over a k-letter alphabet, and $L(A)$ is finite, then there is a regular expression e specifying $L(A)$ with at most $kn(n+1)(n-1)^{(\log n)+1}$ positions.*

Our NFA may accept infinite languages, but the strongly connected components are cliques and can therefore be assumed to consist of a single node. It is obvious that the conversion of an NFA $A = (\Sigma, Q, I, F, \Delta)$ with trivial loops can be reduced in polynomial time to the conversion of acyclic NFAs:

one only needs to compute the symbols S_q that allow to loop on each state q, remove the loops, replace Σ with $\Sigma \times Q$, and replace transitions (q, a, q') with $(q, (a, q'), q')$. One then computes the regular expression for the resulting automaton, and deduces the regular expression for A as follows: for each $b, a \in \Sigma, q \in Q$, one adds $(\bigcup_{a \in S_q} a)^*$ after each occurrence of (b, q) in the regular expression (nothing is added when $S_q = \varnothing$). Finally one replaces every label of the form (b, q) with b. This gives the following result:

Corollary 6.27. *If A is an NFA with n states over a k-letter alphabet, and $L(A)$ is finite, then there is a regular expression e specifying the subwords of $L(A)$ with at most $k(kn)n(n+1)(n-1)^{(\log n)+1} \in 2^{O((\log n)^2)}$ positions.*

We do not have a matching lower bound. However, we exhibit a family of DFAs A_n with n states, such that the size of any regular expression e_n accepting the subwords of $L(A_n)$ cannot be bounded by any polynomial. The proof is a minor adaptation of [EZ74], and is therefore postponed to the Appendix.

Lemma 6.28. *There exist a family of DFAs A_n with size n such that any deterministic expression accepting the subwords of $L(A_n)$ has size $n^{\Omega(\log \log n)}$.*

We plan to investigate whether the techniques of Gruber et al. may help to match the lower and upper bound. In particular, Gruber and Johannsen [GJ08] prove that the conversion of acyclic DFA into regular expressions involves a $n^{\Omega(\log(n))}$ lower bound, matching the $n^{O(\log(n))}$ upper bound of [EKSW05]. But it is not yet clear to us whether the result can be adapted when we wish to represent the subwords of the language. The table in Figure 6.5 summarizes the size of the subword approximation, with rows denoting the format for the input, whereas columns distinguish the format expected for the approximation. The meaning of n depends on the kind of input: when the input is an automaton, n denotes its number of states, and when the input is a regular expression or grammar, n denotes its size. The size of the alphabet is denoted by k.

6.3.2. Indistinguishability of Approximation

We recall that in our opinion the primary purpose of the view schema is to guide the user in her attempt to formulate a meaningful query, and an approximation of the schema should be judged from this perspective. Consequently, we propose the following notion to identify the approximations that prevent the user from formulating unsatisfiable queries.

Definition 6.6. *We say that two sets L_1 and L_2 of Σ-trees are* indistinguishable *by a class $\mathcal{C}$ of queries, denoted $L_1 \approxeq_{\mathcal{C}} L_2$, when every $Q \in \mathcal{C}$ is satisfied by a tree in L_1 if and only if it is satisfied by a tree in L_2.*

	NFA	reg. exp.	DFA	det. reg. exp.
reg. exp	$O(n)$	$O(n)$	$2^{O(n)}$ ¶	$2^{O(n\log k)}$ ¶
NFA	$O(n)$	$2^{O((\log n)^2+\log k)}$ §	$2^{O(n)}$ †	$2^{O(n\log k)}$ †
DFA	$O(n)$	$2^{O((\log n)^2+\log k)}$ §	$2^{O(n)}$ †	$2^{O(n\log k)}$ †
CFG	$2^{O(n)}$ ‡	$2^{O(n)}$ ‡	$2^{2^{O(n)}}$ $\star$	$2^{2^{O(n)}}$ $\star$

*: optimal complexity. The lower bound is obtained from Prop. 6.24.
†: optimal complexity. The lower bound is obtained from [Okh10].
‡: optimal complexity. The lower bound is obtained from Remark 6.5
§: non-polynomial lower bound by Lemma 6.28.
¶: non-polynomial lower bound by Lemma 6.25.

Figure 6.5.: "State" complexity for the subword closure operation.

Given a CDTD H, we denote its subset approximation by H^{Set}, its Parikh approximation by H^{Parikh}, and its subword approximation by H^{Word}.

Theorem 6.29. *For any CDTD H we have*

(i) H and H^{Set} are indistinguishable by $\mathcal{C}_1 = \mathcal{X}Reg(\Downarrow)$.

(ii) H and H^{Parikh} are indistinguishable by $\mathcal{C}_2 = \mathcal{X}Reg(\Downarrow, \Uparrow, [\,], \mathsf{not})$.

(iii) H and H^{Word} are indistinguishable by $\mathcal{C}_3 = \mathcal{X}Reg(\Downarrow, \Uparrow, \Rightarrow^+, \Leftarrow^+, [\,])$.

Proof. (i) We observe that $L(H) \subseteq L(H^{\text{Word}})$, and consequently, it suffices to show that for any query $q \in \mathcal{C}_2$ that is satisfied by a $t \in L(H^{\text{Word}})$ there exists some $t' \in L(H)$ satisfying Q. We remark that, indeed, for every $t \in L(H^{\text{Word}})$ there exists a $t' \in L(H)$ such that $N_t \subseteq N_{t'}$, $root_r = root_{t'}$, $child_t \subseteq child_{t'}$, $lab_t \subseteq lab_{t'}$, and $follow_t \subseteq follow^+_{t'}$. Since queries in $Q \in \mathcal{C}_2$ use neither negation nor horizontal axes (except $\Rightarrow^+, \Leftarrow^+$), adding subtrees under some nodes of t cannot invalidate Q. Consequently, t' satisfies Q.

(ii) It can be shown with an immediate inductive argument that for every $t \in L(H)$ there exists $t' \in L(H^{\text{Parikh}})$ that differs from t only by the relative order of siblings, i.e. $N_{t'} = N_t$, $root_{t'} = root_t$, $lab_{t'} = lab_t$, and $child_t = child_{t'}$. Since the semantics of the queries in $\mathcal{C}_1$ does not depend on $follow_t$, any query $Q \in \mathcal{C}_2$ is satisfied by $t \in L(H)$ if and only if Q is satisfied by the corresponding $t' \in L(H^{\text{Parikh}})$. Similarly, every query satisfied in some tree t' of $L(H^{\text{Parikh}})$ will also be satisfied in some (in any) tree $t \in L(H)$ that differs from t' only by the order of siblings, which concludes the proof.

(iii) By $\mathit{path}(t)$ we denote the set of all descending paths from the root node to any node of t and we extend path to sets of trees in the standard way. We observe, that $L_1 \approx_{\mathcal{C}_3} L_2$ if and only if $\mathit{path}(L_1) = \mathit{path}(L_2)$. Clearly, $\mathit{path}(L(H)) = \mathit{path}(L(H^{\text{Set}}))$. □

We also remark that the subword and the subset methods construct a superset of the real schema. More precisely, $L(H) \subseteq H^{\text{Word}} \subseteq H^{\text{Set}}$. As for Parikh approximation, H^{Parikh} correctly characterizes H if we consider unordered trees. We now present a proof that XPath dialects allowing other combinations of horizontal axes cannot be approximated with DTDs.

Approximability and "Optimality" of our Approximations

Proposition 6.30. *There exists a CDTD H from which no DTD is indistinguishable by $\mathcal{C}_0 = \mathcal{X}Reg(\Downarrow, \Rightarrow^+, [\,], \mathsf{not})$ or $\mathcal{C}_0' = \mathcal{X}Reg(\Downarrow, \Rightarrow)$.*

Proof. For $\mathcal{C}_0$ we observe that for every tree there exists a query in $\mathcal{C}_0$ that is satisfied by that tree and isomorphic trees only, i.e., this query characterizes the tree up to isomorphism. For example, for $\mathtt{r}(\mathsf{a}, \mathsf{b})$ the query can be expressed as:

$$\mathsf{self}::\mathtt{r}\big[\Downarrow::\mathsf{a}[\,\mathsf{not}(\Downarrow)]/\Rightarrow^+::\mathsf{b}[\,\mathsf{not}(\Downarrow)\ \mathsf{and}\ \ \mathsf{not}(\Rightarrow^+)]\,\mathsf{and}\ \ \mathsf{not}(\Downarrow/\Rightarrow^+::\mathsf{a}/\Rightarrow^+::\mathsf{b})\big].$$

Consequently $L_1 \approx_{\mathcal{C}} L_2$ iff $L_1 \equiv L_2$ for any $\mathcal{C}$ containing $\mathcal{C}_0$.

The proof for $\mathcal{C}_0'$ is a bit more intricate. Let H be a CDTD such that $L(H) = \{\mathtt{r}(\mathsf{c}, \mathsf{a}^k, \mathsf{b}^k, \mathsf{c}) \mid k \in \mathbb{N}\}$ and assume that there is a DTD D such that $H \approx_{\mathcal{C}_0'} D$.

We observe that $L(D)$ consists of trees of depth 1 since $\Downarrow::*/\Downarrow::*$ is not satisfied by any tree in $L(H)$. Also,

$$D(\mathtt{r}) \subseteq (\epsilon + \mathsf{c})\mathsf{a}^*\mathsf{b}^*(\epsilon + \mathsf{c}) \tag{6.1}$$

since no tree in $L(H)$ satisfies any of the queries

$$\begin{aligned} &\mathsf{self}::\mathtt{r}/\Downarrow::*/\Rightarrow^+::\mathsf{c}/\Rightarrow^+::\mathsf{c}, \qquad && \mathsf{self}::\mathtt{r}/\Downarrow::\mathsf{c}/\Rightarrow^+::\mathsf{c}/\Rightarrow^+::*,\\ &\mathsf{self}::\mathtt{r}/\Downarrow::\mathsf{a}/\Rightarrow^+::\mathsf{b}/\Rightarrow^+::\mathsf{a}, && \mathsf{self}::\mathtt{r}/\Downarrow::\mathsf{b}/\Rightarrow^+::\mathsf{a}/\Rightarrow^+::\mathsf{b}. \end{aligned}$$

Define the following objects

$$\begin{aligned} R_1 &= L(H(\mathtt{r})) = \{\mathsf{c}\mathsf{a}^k\mathsf{b}^k\mathsf{c} \mid k \in \mathbb{N}\},\\ R_2 &= L(D(\mathtt{r})) \cap L(\mathsf{c}\mathsf{a}^*\mathsf{b}^*\mathsf{c}),\\ Q_{n,m} &= \mathsf{self}::\mathtt{r}/\Downarrow::\mathsf{c}/(\Rightarrow::\mathsf{a})^n/(\Rightarrow::\mathsf{b})^m/\Rightarrow::\mathsf{c}, \end{aligned}$$

and note that R_2 is regular (being an intersection of two regular sets).

Given (6.1), $H \approx_{\mathcal{C}_0} D$ with $Q_{k,k}$ for $k \in \mathbb{N}$ implies $R_1 \subseteq R_2$, since for all $k \in \mathbb{N}$ the query $Q_{k,k}$ is satisfied by a tree in $L(H)$. In a similar way, we can show that for every $w = \mathtt{c}\mathtt{a}^{k_1}\mathtt{b}^{k_2}\mathtt{c} \in R_2$ we have $k_1 = k_2$, i.e. $w \in R_1$. Indeed, if there was some $w = \mathtt{c}\mathtt{a}^{k_1}\mathtt{b}^{k_2}\mathtt{c} \in R_2$ with $k_1 \neq k_2$, the query Q_{k_1,k_2} would be satisfied on D, and thus on H, which is not the case. Consequently, $R_2 = R_1$ is not a regular set, a contradiction. □

Theorem 6.31. *Take any class of queries $\mathcal{C}$ containing $\mathcal{C}_4 = \mathcal{X}Reg(\Downarrow, [\,])$ or $\mathcal{C}_4' = \mathcal{X}Reg(\Downarrow, \Uparrow)$. For any $k \in \mathbb{N}$ there exists a CDTD H_k such that $|H_k| = O(k)$ and for any DTD D indistinguishable from H_k by $\mathcal{C}$ the size of D is $\Omega(2^k)$.*

Proof. We consider the CDTD H_k such that

$$\mathtt{r} \to \mathtt{a}^{2^k} \qquad\qquad \mathtt{a} \to \mathtt{b} \qquad\qquad \mathtt{b} \to \mathtt{b} \mid \mathtt{c}$$

Clearly, H_k can be constructed in a manner such that $|H_k| = O(k)$ (see D_3 in Example 6.1). Now, let D be any DTD indistinguishable from H_k by $\mathcal{C}_4$. It can be easily shown that

$$\begin{aligned} D(\mathtt{c}) &= \epsilon, & \mathtt{b} + \mathtt{c} \subseteq & D(\mathtt{b}) \subseteq \mathtt{b} + \mathtt{c}^*, \\ D(\mathtt{r}) &\subseteq \mathtt{a}^*, & \mathtt{b} \subseteq & D(\mathtt{a}) \subseteq b + \epsilon, \end{aligned}$$

We claim that: (i) $\mathtt{a}^{2^k} \in L(D(\mathtt{r}))$, and (ii) $L(D(\mathtt{r})) \subseteq \{\mathtt{a}^m \mid 0 \leqslant m \leqslant 2^k\}$. For (i) it suffices to consider the query $\mathsf{self}::\mathtt{r}/Q_1/\cdots/Q_{2^k}$, where

$$Q_i = \mathsf{self}::*[\Downarrow::\mathtt{a}/(\Downarrow::\mathtt{b})^i/\Downarrow::\mathtt{c}].$$

To show (ii) consider the query $\mathsf{self}::\mathtt{r}/Q_1/\cdots/Q_{2^k}$ for any $m > 2^k$. It is not satisfied by any tree in $L(H_k)$ and so it cannot be satisfied by any tree in $L(D)$. Since $L(D(r))$ is a set of words whose length is bounded by 2^k, then the length of the regular expression $D(r)$ must be at least 2^k. We prove the lower bound for $\mathcal{C}_4'$ with the same argument but using $Q_i = \Downarrow::\mathtt{a}/(\Downarrow::\mathtt{b})^i/\Downarrow::\mathtt{c}/(\Uparrow::*)^{i+2}$. □

Figure 6.6 summarizes our results on approximations.

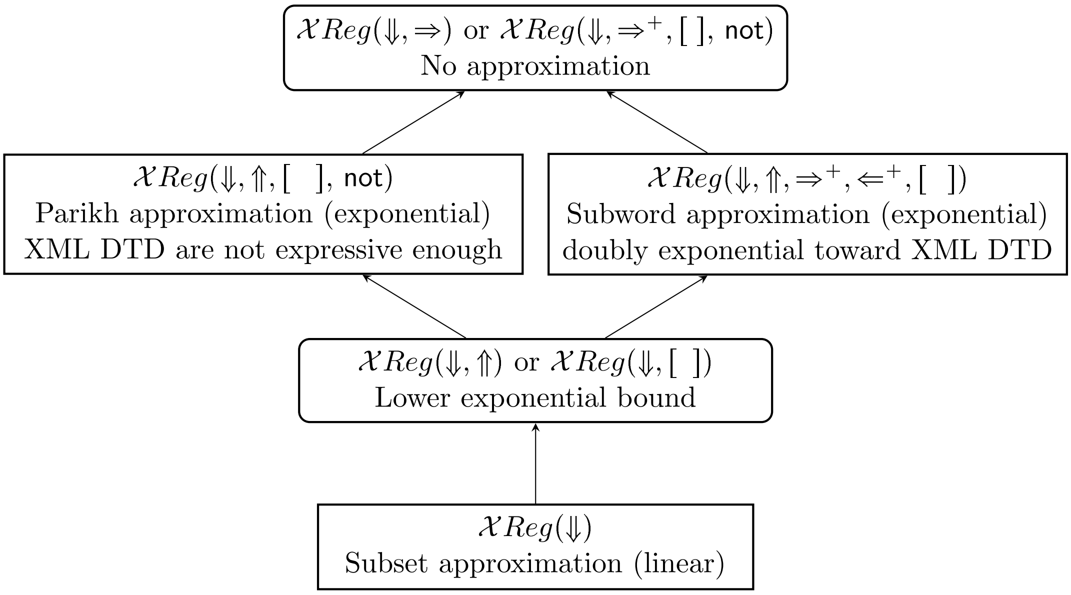

$\mathcal{X}Reg(\Downarrow, \Rightarrow)$ or $\mathcal{X}Reg(\Downarrow, \Rightarrow^+, [\,], \mathsf{not})$
No approximation
$\mathcal{X}Reg(\Downarrow, \Uparrow, [\quad], \mathsf{not})$
Parikh approximation (exponential)
XML DTD are not expressive enough
$\mathcal{X}Reg(\Downarrow, \Uparrow, \Rightarrow^+, \Leftarrow^+, [\quad])$
Subword approximation (exponential)
doubly exponential toward XML DTD
$\mathcal{X}Reg(\Downarrow, \Uparrow)$ or $\mathcal{X}Reg(\Downarrow, [\quad])$
Lower exponential bound
$\mathcal{X}Reg(\Downarrow)$
Subset approximation (linear)

Figure 6.6.: Summary of approximation results (negative results in round boxes).

Conclusion

> What we call the beginning is often the end
> And to make an end is to make a beginning.
> The end is where we start from.
>
> *(T.S. Elliot, Little Gidding)*

The purpose of this thesis was to develop new tools for (non-materialized) view based access control over XML documents. We investigated which problems could be solved using formal methods from automata and language theory. As illustrated in this thesis, query rewriting methods provide numerous mechanisms to execute or restrict query and update operations on XML data in this framework, with polynomial query complexity and linear data complexity. We also developed new algorithms to check properties of policies, such as determinacy or applicability of view updates under some restrictions on authorized updates.

After introducing general algorithms and results about word and tree languages, we present our framework for non-materialized security views. The more technical contributions focus on reasoning about evolving policies and documents, as well as techniques for providing a view schema to the user.

Summary of the Contributions

General algorithms A first contribution of this dissertation is a detailed survey of algorithms for membership and evaluation for visibly pushdown automata, together with a few insights on the conversion between VPAs and other tree automata models. We also define tree alignments as a unified model for views, queries and updates. In the chapter about updates, we study the behaviour of tree alignment languages with respect to the join and composition operations.

Access control model The cornerstone of this thesis is the definition of an access control model inspired by the non-materialized view-based framework of [FCG04, FGJK07, KMR09]. Compared to these previous models, we can use a more larger query language for the view definition and the user queries. As in the previous models, we rewrite the users query before they are evaluated. Our choice of using $\mathcal{X}Reg$ allows to simplify the previous query rewriting algorithms: in particular, the addition of upward axes greatly simplifies the rewriting algorithm in comparison to [FGJK07]. The asymptotic complexity of query evaluation remains similar to the complexity of the previous models. However, it is not clear whether the optimization techniques developed in [FGJK07] can be adapted to our model.

Due to the higher expressiveness of our views, it becomes harder to derive a schema for the view and to reason about the policy. We therefore introduce three restrictions that facilitate the derivation of a view schema, the comparison of policies, and the processing of updates. Two of these restrictions, upward-closed views [MTKH06, DFGM08, LS10, LLLL11], and bounded depth documents [FCG04, KMR05, BCF07, BFG08], are classical restrictions when dealing with tree languages. The third one: interval-boundedness, seems to us a natural generalization of the other two.

Policy comparison We believe that tools for evaluating which information can be obtained from a view can prove useful to a database administrator. In particular, the administrator may wish to check if a modification of the policy does not disclose some information that was previously hidden. There are many possible criteria for comparing security views. The most intuitive definition just checks containment of the views. As this comparison does not capture precisely the information that can be obtained from the policies, we propose two further comparisons. The first one compares policies in terms of what (unary) queries can be expressed on the view. We show that this comparison can be expressed in terms of a query rewriting (or determinacy) problem. This problem is undecidable in general, but can be decided for interval-bounded views, though with exponential complexity. The comparison becomes tractable under tighter restrictions. The other comparison contrasts the certain answers for both views, and can be evaluated with complexity similar to the second comparison. Under very general assumptions on the view language, containment can be reduced in the second comparison, which in turns can be reduced to the third. In terms of expressiveness, the second comparison refines the third one and the containment, whereas the containment and the third comparison are incomparable.

View update translation Support of update operations is a crucial feature in database systems. We investigate the view update problem: given an update (function) that the user wishes to execute on its view, we compute the corresponding update (function) that must be executed on the source document. We consider two cases: in the unconstrained case, every update that maps a source document to another source document (satisfying the schema) is authorized, whereas in the constrained setting only a subset of these updates are allowed. The constrained setting raises interesting questions, such as deciding uniform translatability. We introduce k-synchronized updates, a restriction on the updates which makes uniform translatability decidable and allows to solve the view update problem in the constrained case.

View schema derivation The view schema obtained from a general view needs not be regular. For interval-bounded views, however, it is regular, but still needs not be a DTD. We propose three techniques to approximate the view schema with a DTD: the Subset, Subword and Parikh approximations. The Subword and Parikh approximations refine the subset approximations, and capture more information about the schema, but on the other hand the productions of the resulting DTD have exponential size, whereas the Subset approximation can be computed in linear time. Furthermore, the Subset approximation is an XML DTD, whereas some view schemata do not admit Parikh approximations with deterministic productions. The Subword approximation can be modified so that its output is an XML DTD, but in that case the resulting productions are doubly exponential.

It is undecidable whether a general view schema (defined with context-free DTDs) can be defined with a DTD, an XML DTD, or a tree automaton. However, if the view schema is regular, as is the case for interval-bounded views, testing if it can be defined with a DTD is EXPTIME-complete [MNSB06], and similarly for XML DTDs. When the view schema is given as a DTD, one can check if it can be defined with an XML DTD in exponential time, and the problem is PSPACE-hard [BGMN09]. In contrast with these exponential complexities, we provide a new algorithm that tests if a given DTD is an XML DTD in linear time, whereas existing algorithms had quadratic complexity.

Further directions of study

Increasing the expressivity of views and queries The views and query languages in this dissertation are restricted to the navigational core of XPath, or use tree automata. This may appear too restrictive for practical applications, which may require to support key mechanisms for the view, and data aggregation for the queries. One may consider extending the framework to address this shortcoming, using data logic for instance. The view language could also be extended to allow restructuring the document. Several transducer models for unranked trees, such as [AD12] could be considered to define views that copy and reorganize parts of a document.

From trees to graphs Native XML databases remain the exception rather than the norm. This may limit the use of access control models for tree-structured documents. And indeed, we observe on Figure 6.7 a decline in the community's interest for XML access control, while access control in general remains an active topic of research. However, query rewriting techniques could find applications for graph-structured data, to query ontologies for instance. This has already been investigated in the literature for various query languages such as conjunctive queries to Datalog and regular path queries, but could find new applications with the recent proposal of SPARQL as a query language.

Optimizations for VPAs Visibly pushdown automata have raised increasing interest since their introduction by Alur and Madhusudan [AM04b, AM09], as evidenced on Figure 6.7. Yet we believe that many fundamental issues over VPAs are not fully understood yet. Optimizing the evaluation of VPAs (and especially of extensions of VPAs defining queries) can still be considered a research topic. These questions pertain to the problem of handling efficiently non determinism in different automata and transducer models. Non-determinism is particular challenging for VPAs, and is problematic for most tree automata models. Also, we are not aware of much work that would address (directly) the efficient evaluation of non-deterministic transducers.

Optimizations for (XML) Schema languages The opportunity of the determinism constraint for regular expressions in XML DTD, XML Schema and SGML has been debated [Man01, W3C]. Current algorithms do not fully exploit the determinism requirement to gain performance, but it is not clear whether our algorithms for deterministic regular expressions would allow substantial optimizations for schemata of reasonable size. We plan to compare experimentally our algorithms with state of the art parsers and regular expression libraries... These algorithms also raise numerous other questions, on the precise complexity of evaluation, containment, and equivalence of deterministic regular expressions. Some other interleaving and shuffling operators from classical schema languages should also be considered. Furthermore, one may wonder whether the techniques developed can be used for regular expressions occurring in other contexts.

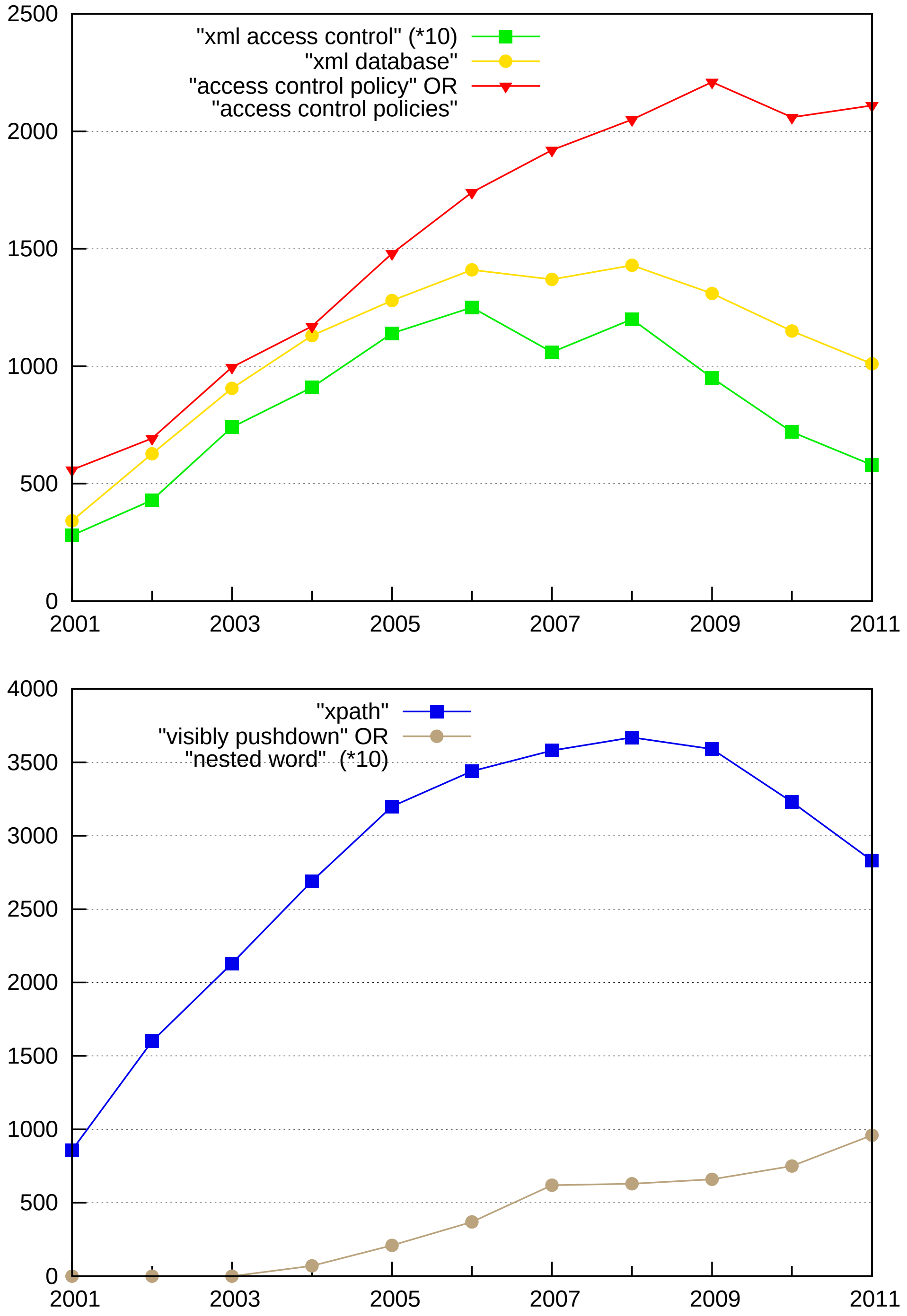


Figure 6.7.: Google scholar results by domain: 2001-2011

The results in Figure 6.7 have been obtained from queries on Google scholar on June 6th, 2012 (including citations and patents). The accurate numbers are not very relevant, but can be considered as an indicator on general trends in the research community. Nota Bene: the numbers for the two queries 1) "XML access control" and 2) "visibly pushdown" OR "nested word" have been scaled by a factor of 10 for better readability.

Notations

Notation	Description	Def. on page
Words and Trees		
w : `word`, a : `letter`, t : `tree`, i : `natural`		
$\|w\|$	size of the word w	48
$\|w\|_a$	number of occurrences of a in w	48
$w[i]$	i^{th} letter in w	48
$\odot$	concatenation symbol	52
T_Σ	set of all trees over Σ	48
Σ_t	alphabet of t	48
follow_t	following sibling predicate	48
child_t	child predicate	48
root_t	root	48
Parent_t	parent predicate	49
next_t	next sibling predicate	49
$\mathit{fcns}(t)$	first child next sibling encoding	50
$\mathit{yield}(t)$	yield	58
$\rho^\uparrow$	run	62
ω	the exponent in the complexity of matrix multiplication	47
$\simeq$	isomorphism relation	50
Regular expressions		
e : `regular expression`, n : `position`, i, j : `natural`		
$\mathit{Pos}(e)$	positions	52
$e^{[i..j]}$	repetition of e	53
$\mathit{First}(e)$	first positions	55
$\mathit{Last}(e)$	last positions	55
$\mathrm{Glushkov}(e)$	Glushkov automaton	55
$\mathit{Follow}_e(n)$	following positions	55
Tree automata		
$\mathcal{A}$: `NTA` *or* `VPA`, q : `state`		
$\mathcal{A}_q$	language accepted below q (NTA)	59
$\mathcal{A}_{q,q'}$	language accepted from q to q' (VPA)	62

Notation	Description	Def. on page
$Acc_{\mathcal{A}}$	horizontal accessibility relation of $\mathcal{A}$	62

Queries and Views

Q, Q_v : `query`, $\mathcal{C}, \mathcal{C}'$: `class of queries`, t : `tree`, D : `DTD`, `ann` : `annotation`, $\mathcal{X}$: `XPath query`

$\mathrm{dom}(Q)$	domain	87
$Q\,(t)$	Answer of query	89
$Memb(\mathcal{C}, \mathcal{C}')$	membership problem	93
$\Sigma_{\mathsf{edit},k}$	alphabet for alignments	96
$\pi_{i_1,i_2,\ldots,i_m}$	projection	96
$t \otimes Q$	annotation of a tree	97
$\mathcal{V}iew\,(Q_v, t)$	view tree	98
(D, ann)	annotated DTD	116
$\preccurlyeq_1$	comparison by inclusion	129
$\preccurlyeq_2$	comparison by determinacy	132
$\preccurlyeq_3$	determinacy modulo isomorphism	132
$\preccurlyeq_{2,\mathcal{C}}$	comparison by query rewriting	131
$Certain_{Q_v}\,(Q; t_v)$	certain answers	133
$\mathrm{Ant}\,(t, Q_v)$	view inverse	133
$Q_{(D,\mathsf{ann})}$	query defined by annotated DTD	116
$\mathcal{X}Reg$	Regular XPath language	90
$Q_{\mathcal{X}}$	query defined by $\mathcal{X}$	91
$Filt(Q)$	filter corresponding to Q	91
$\mathcal{X}^{-1}$	inverse expression	91

Updates

u, u', u_v : `editing script`, V : `view`, $L, \mathcal{U}_s$: `set of editing scripts`

$\sim$	equivalence of editing scripts	165
$[u]$	equivalence class of u	165
$\Phi_1(u), \Phi_2(u)$	morphisms for equivalence	165
u^{-1}	inverse of u	166
$u \bowtie u'$	synchronization of editing scripts	166
$u \circ u'$	composition of editing scripts	167
$\mathrm{Prop}\,(V, u_v)$	propagations of u_v w.r.t. V	174
$Unif\,(V, \mathcal{U}_s)$	uniform edit. scripts (w.r.t. V and $\mathcal{U}_s$)	184
$Sync(k, L)$	k-synchronized scripts of L	189
$\mathcal{U}_V^k$	edit. scripts inducing k-sync. ed. scripts	191

Index

List of Figures

Bibliography

> Stella: We've become a race of Peeping Toms. What people ought to do is get outside their own house and look in for a change. Yes sir. How's that for a bit of homespun philosophy?
> Jeff: Readers Digest, April 1939.
> Stella: Well, I only quote from the best.
>
> *(Hitchcock, Rear window)*

[AAC$^+$99] Serge Abiteboul, Bernd Amann, Sophie Cluet, Anat Eyal, Laurent Mignet, and Tova Milo. Active views for electronic commerce. In *VLDB*, pages 138–149, 1999. (Cited page 36)

[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 pages 150, 278, and 279)

[ABG$^+$05] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. In *CIDR*, pages 186–199, 2005. (Cited page 20)

[ABGA11] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Emilien Antoine. A rule-based language for web data management. In *PODS*, pages 293–304, 2011. (Cited page 20)

[ABM08] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008. (Cited page 20)

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, pages 87–98, 2007. (Cited page 25)

[ABQ09] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS*, pages 107–123, 2009. (Cited page 223)

[AC11] Foto N. Afrati and Rada Chirkova. Selecting and using views to compute aggregate queries. *J. Comput. Syst. Sci.*, 77(6):1079–1107, 2011. (Cited page 22)

[ACG+09] Foto N. Afrati, Rada Chirkova, Manolis Gergatsoulis, Benny Kimelfeld, Vassia Pavlaki, and Yehoshua Sagiv. On rewriting XPath queries using views. In *EDBT*, pages 168–179, 2009. (Cited page 24)

[AD12] Rajeev Alur and Loris D'Antoni. Streaming tree transducers, *to appear at ICALP (2)*, 2012. (Cited page 235)

[Afr11] Foto N. Afrati. Determinacy and query rewriting for conjunctive queries and views. *Theor. Comput. Sci.*, 412(11):1005–1021, 2011. (Cited page 23)

[AGKR04] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37:441–456, 2004. 10.1007/s00224-004-1155-5. (Cited page 205)

[AH87] I. J. Aalbersberg and H. J. Hoogeboom. Decision problems for regular trace languages. In *14th International Colloquium on Automata, languages and programming*, pages 250–259, 1987. (Cited page 165)

[AI00] Natasha Alechina and Neil Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337, 2000. (Cited pages 92 and 127)

[AJREF10] Ryma Abassi, Florent Jacquemard, Michael Rusinowitch, and Sihem Guemara El Fatmi. XML Access Control: from XACML to Annotated Schemas. In *Second International Conference on Communications and Networking (ComNet)*, pages 1–8, Tozeur, Tunisie, 2010. IEEE Computer Society Press. (Cited page 12)

[AKG91] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187, 1991. (Cited page 26)

[AL05] Marcelo Arenas and Leonid Libkin. An information-theoretic approach to normal forms for relational and XML data. *J. ACM*, 52(2):246–283, 2005. (Cited page 32)

[Alu07] Rajeev Alur. Marrying words and trees. In *PODS*, pages 233–242, 2007. (Cited page 60)

[AM04a] Cyril Allauzen and Mehryar Mohri. An optimal pre-determinization algorithm for weighted transducers. *Theor. Comput. Sci.*, 328(1-2):3–18, 2004. (Cited page 40)

[AM04b] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004. (Cited pages iv, 60, 79, and 236)

[AM09] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009. (Cited pages 60, 63, 64, 66, 79, and 236)

[AMR$^+$98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998. (Cited page 28)

[AP09] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *ICDT*, pages 121–126, 2009. (Cited page 48)

[ASV06] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and querying XML with incomplete information. *ACM Trans. Database Syst.*, 31(1):208–254, 2006. (Cited page 27)

[AU71] Alfred V. Aho and Jeffrey D. Ullman. Translations on a context-free grammar. *Information and Control*, 19(5):439–475, 1971. (Cited page 67)

[B6̈0] A. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math*, 6:66–92, 1960. (Cited page 139)

[BBFV05] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Verification of tree updates for optimization. In *CAV*, pages 379–393, 2005. (Cited page 30)

[BC02] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over finite and infinite words. *Theor. Comput. Sci.*, 289(1):225–251, 2002. (Cited page 40)

[BC06] Mikolaj Bojanczyk and Thomas Colcombet. Tree-walking automata cannot be determinized. *Theor. Comput. Sci.*, 350(2-3):164–173, 2006. (Cited page 67)

[BC08] Mikolaj Bojanczyk and Thomas Colcombet. Tree-walking automata do not recognize all regular languages. *SIAM J. Comput.*, 38(2):658–701, 2008. (Cited page 67)

[BC09] Michael Benedikt and James Cheney. Schema-based independence analysis for XML updates. *PVLDB*, 2(1):61–72, 2009. (Cited pages 29 and 30)

[BC10] Michael Benedikt and James Cheney. Destabilizers and independence of XML updates. *PVLDB*, 3(1):906–917, 2010. (Cited page 31)

[BCF07] Loreto Bravo, James Cheney, and Irini Fundulaki. Repairing inconsistent XML write-access control policies. In *DBPL*, pages 97–111, 2007. (Cited pages 39 and 234)

[BCF08] Loreto Bravo, James Cheney, and Irini Fundulaki. ACCOn: checking consistency of XML write-access control policies. In *EDBT*, pages 715–719, 2008. (Cited page 39)

[BCF$^+$10] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *ICFP*, pages 193–204, 2010. (Cited page 36)

[BDH06] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. PATAXO: A framework to allow updates through XML views. *ACM Transactions on Database Systems (TODS)*, 31, 2006. (Cited page 37)

[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal logic*. Cambridge University Press, New York, NY, USA, 2001. (Cited pages 150, 278, and 279)

[Bel05] David Elliott Bell. Looking back at the Bell-La Padula model. In *ACSAC*, pages 337–351, 2005. (Cited page 2)

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997. (Cited page 73)

[Ber79] Jean Berstel. *Transductions and context-free languages*, volume 38 of *Leitfäden der Angewandten Mathematik und Mechanik*. B. G. Teubner, Stuttgart, 1979. (Cited page 185)

[BF05] Michael Benedikt and Irini Fundulaki. XML Subtree Queries: Specification and Composition. In *DBPL*, pages 138–153, 2005. (Cited page 21)

[BF11] Mikolaj Bojanczyk and Diego Figueira. Efficient evaluation for a temporal logic on changing XML documents. In *PODS*, pages 259–270, 2011. (Cited page 37)

[BFCP$^+$05] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005. (Cited page 205)

[BFFN05] Philip Bohannon, Wenfei Fan, Michael Flaster, and P. P. S. Narayan. Information preserving XML Schema embedding. In *VLDB*, pages 85–96, 2005. (Cited page 26)

[BFG$^+$06] Peter A. Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, K. Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery-consistent and efficient updates on the pre/post plane. In *EDBT*, pages 1190–1193, 2006. (Cited page 40)

[BFG08] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008. (Cited pages 94 and 234)

[BGM10] Henrik Björklund, Wouter Gelade, and Wim Martens. Incremental XPath evaluation. *ACM Trans. Database Syst.*, 35(4):29, 2010. (Cited pages 28, 29, and 111)

[BGMN09] Geert Jan Bex, Wouter Gelade, Wim Martens, and Frank Neven. Simplifying XML schema: effortless handling of nondeterministic regular expressions. In *SIGMOD Conference*, pages 731–744, 2009. (Cited pages 42, 202, 218, 219, 235, and 291)

[BGMS11] Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. In *EDBT*, pages 177–188, 2011. (Cited pages 28 and 29)

[BGNV10] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010. (Cited pages 41 and 42)

[BGT$^+$11] I. Boneva, B. Groz, S. Tison, A.-C. Caron, Y. Roos, and S. Staworko. View update translation for XML. In *International Conference on Database Theory (ICDT)*, pages 42–53. ACM, March 2011. (Cited page 9)

[BGvK$^+$06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006. (Cited page 37)

[BJ07] Michael Benedikt and Alan Jeffrey. Efficient and expressive tree filters. In *FSTTCS*, pages 461–472, 2007. (Cited page 114)

[BK93] A. Brüggemann-Klein. Regular expressions into finite automata. *TCS*, 120(2):197–213, 1993. (Cited page 205)

[BK08] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008. (Cited pages 46, 90, and 93)

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets (version 1). Unpublished manuscript, 2001. (Cited pages 66 and 99)

[BKW98] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253, 1998. (Cited pages 203, 217, 218, 219, 224, and 291)

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973. (Cited page 2)

[BLPS09] Pablo Barceló, Leonid Libkin, Antonella Poggi, and Cristina Sirangelo. XML with incomplete information: models, properties, and query answering. In *PODS*, pages 237–246, 2009. (Cited pages 26 and 27)

[BM63] J. A. Brzozowski and E. J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *Electronic Computers, IEEE Transactions on*, EC-12(2):67 –76, april 1963. (Cited page 54)

[BMV06] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 9(2):187–212, 2006. (Cited pages vii, 44, and 77)

[BN89] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989. (Cited page 18)

[BNdB04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: A Practical Study. In *WebDB*, pages 79–84, 2004. (Cited page 44)

[BNSV10] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2), 2010. (Cited pages 41 and 204)

[BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML Schema Definitions from XML Data. In *VLDB*, pages 998–1009, 2007. (Cited page 41)

[BÖB$^+$04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, pages 60–71, 2004. (Cited page 25)

[Boj08] Mikolaj Bojanczyk. Tree-walking automata. In *LATA*, pages 1–2, 2008. (Cited page 67)

[BP10] Changwoo Byun and Seog Park. A schema based approach to valid XML access control. *J. Inf. Sci. Eng.*, 26(5):1719–1739, 2010. (Cited page 15)

[BP11] Mikołaj Bojańczyk and Pawel Parys. XPath evaluation in linear time. *J. ACM*, 58(4):17, 2011. (Cited pages 68, 204, 210, 215, and 290)

[BPS96] Graham Brightwell, Hans Jürgen Prömel, and Angelika Steger. The average number of linear extensions of a partial order. *J. Comb. Theory, Ser. A*, 73(2):193–206, 1996. (Cited page 110)

[BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004. (Cited page 28)

[BS81] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981. (Cited pages 33, 34, and 184)

[BS09] Michael Benedikt and Luc Segoufin. Regular tree languages definable in FO and in FO_{mod}. *ACM Trans. Comput. Log.*, 11(1), 2009. (Cited page 94)

[BTCU12] Nicole Bidoit-Tollu, Dario Colazzo, and Federico Ulliana. Type-based detection of XML query-update independence. *PVLDB*, 5(9):872–883, 2012. (Cited page 31)

[BW08] Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In *Automata and Logic: History and Perspectives. Collected papers for Wolfgang Thomas' 60th birthday*, 2008. (Cited page 51)

[CAM09] Bogdan Cautis, Serge Abiteboul, and Tova Milo. Reasoning about XML update constraints. *J. Comput. Syst. Sci.*, 75(6):336–358, 2009. (Cited page 37)

[CCBS05] Frédéric Cuppens, Nora Cuppens-Boulahia, and Thierry Sans. Protection of relationships in XML documents with the XML-BB model. In *ICISS*, pages 148–163, 2005. (Cited page 18)

[CCF$^+$09] James Cheney, Stephen Chong, Nate Foster, Margo I. Seltzer, and Stijn Vansummeren. Provenance: a future history. In *OOPSLA Companion*, pages 957–964, 2009. (Cited page 38)

[CCFV08] Byron Choi, Gao Cong, Wenfei Fan, and Stratis D. Viglas. Updating recursive XML views of relations. *Journal of Computer Science and Technology*, 23, 2008. (Cited page 37)

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007. (Cited pages 51 and 66)

[CDO08] Bogdan Cautis, Alin Deutsch, and Nicola Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008. (Cited page 25)

[CDOV11] Bogdan Cautis, Alin Deutsch, Nicola Onose, and Vasilis Vassalos. Querying XML data sources that export very large sets of views. *ACM Trans. Database Syst.*, 36(1):5, 2011. (Cited page 25)

[CdVF$^+$10] Valentina Ciriani, Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010. (Cited page 20)

[CFG$^+$11] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1, 2011. (Cited page 38)

[CG99] Christian Choffrut and Serge Grigorieff. Uniformization of rational relations. In *Jewels are Forever*, pages 59–71, 1999. (Cited page 181)

[CGLV02] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Lossless regular views. In *PODS*, pages 247–258, 2002. (Cited pages viii and 23)

[CGLV07] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing: On the relationship between rewriting, answering and losslessness. *Theor. Comput. Sci.*, 371(3):169–182, 2007. (Cited pages viii and 23)

[CGLV09] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. An automata-theoretic approach to regular XPath. In *DBPL*, pages 18–35, 2009. (Cited pages 67, 68, 92, 105, 107, 108, 109, 110, 111, 123, 127, 151, and 162)

[CGLV10] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Node selection query languages for trees. In *AAAI*, 2010. (Cited pages 67 and 68)

[CGM11] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Dynamic reasoning on XML updates. In *EDBT*, pages 165–176, 2011. (Cited page 40)

[CH91] Sang Cho and Dung T. Huynh. Finite-automaton aperiodicity is PSPACE-Complete. *Theor. Comput. Sci.*, 88(1):99–116, 1991. (Cited page 94)

[Che08] James Cheney. FLUX: functional updates for XML. In *ICFP*, pages 3–14, 2008. (Cited page 27)

[Che11] James Cheney. A formal framework for provenance security. In *CSF*, pages 281–293, 2011. (Cited page 38)

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959. (Cited page 57)

[Cho77] Christian Choffrut. Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. *Theor. Comput. Sci.*, 5(3):325–337, 1977. (Cited page 40)

[Cho02] Byron Choi. What are real DTDs like? In *WebDB*, pages 43–48, 2002. (Cited page 44)

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. (Cited pages 112 and 114)

[Cla02] newsgroup post, 2002. http://www.imc.org/ietf-xml-use/mail-archive/msg00217.html. (Cited page 87)

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977. (Cited page 47)

[CNS99] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999. (Cited page 22)

[CNT04] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In *RTA*, pages 105–118, 2004. (Cited page 66)

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. (Cited page 32)

[Cou91] Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991. (Cited pages 223 and 224)

[CP84] Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of relational views. *J. ACM*, 31(4):742–760, 1984. (Cited page 34)

[CP97] C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. *TCS*, 178(1–2):1–36, 1997. (Cited pages 56, 206, and 207)

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990. (Cited page 47)

[DB82] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982. (Cited page 33)

[DBI] Verizon 2011 Data Breach Investigation Report, available at: www.verizonbusiness.com/go/2011dbir. (Cited page 4)

[DdVPS02] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for XML documents. *ACM Trans. Inf. Syst. Secur.*, 5(2):169–202, 2002. (Cited pages 13, 16, and 17)

[DFGM08] Ernesto Damiani, Majirus Fansi, Alban Gabillon, and Stefania Marrara. A general approach to securely querying XML. *Computer Standards & Interfaces*, 30(6):379–389, 2008. (Cited pages 12, 18, 19, and 234)

[DFK06] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, pages 632–646, 2006. (Cited page 39)

[DGN$^+$12] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. Early XPath Node Selection on XML Streams. Research report, March 2012. (Cited page 80)

[DKKdO07] Daniel J. Dougherty, Claude Kirchner, Hélène Kirchner, and Anderson Santana de Oliveira. Modular access control via strategic rewriting. In *ESORICS*, pages 578–593, 2007. (Cited page 39)

[DKM$^+$11] Susan B. Davidson, Sanjeev Khanna, Tova Milo, Debmalya Panigrahi, and Sudeepa Roy. Provenance views for module privacy. In *PODS*, pages 175–186, 2011. (Cited page 38)

[DLM10] Claire David, Leonid Libkin, and Filip Murlak. Certain answers for XML queries. In *PODS*, pages 191–202, 2010. (Cited pages 26 and 27)

[Don65] J. E. Doner. Decidability of the weak Second-Order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965. (Cited page 59)

[Don70] John Doner. Tree acceptors and some of their applications. *J. Comput. Syst. Sci.*, 4(5):406–451, 1970. (Cited page 59)

[DTR] Evan Driscoll, Aditya Thakur, and Thomas Reps. OpenNWA, a nested-word-automaton library. http://research.cs.wisc.edu/wpis/OpenNWA, retrieved 02/2012. (Cited page 79)

[dVFJ$^+$10] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Encryption policies for regulating access to outsourced data. *ACM Trans. Database Syst.*, 35(2), 2010. (Cited page 20)

[EGKL11] Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. Parikhs theorem: A simple and direct automaton construction. *Inf. Process. Lett.*, 111(12):614–619, 2011. (Cited pages 221 and 222)

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000. (Cited page 73)

[EKSW05] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-wei Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005. (Cited pages 227 and 228)

[EUd12] European data protection reform. http://ec.europa.eu/justice/newsroom/data-protection/news/120125_en.html, 2012. (Cited page 3)

[EZ74] Andrzej Ehrenfeucht and H. Paul Zeiger. Complexity measures for regular expressions. In *STOC*, pages 75–79, 1974. (Cited pages 54, 228, 292, and 293)

[FCB07] Wenfei Fan, Gao Cong, and Philip Bohannon. Querying XML with update syntax. In *SIGMOD Conference*, pages 293–304, 2007. (Cited page 32)

[FCG04] W. Fan, C.-Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598, 2004. (Cited pages ii, iii, vi, 5, 13, 16, 17, 21, 22, 39, 116, 124, 128, 129, 233, and 234)

[FDL11] Nadime Francis, Claire David, and Leonid Libkin. A direct translation from XPath to nondeterministic automata. In *AMW*, 2011. (Cited pages 100 and 112)

[Feg10] Leonidas Fegaras. Propagating updates through XML views using lineage tracing. In *ICDE*, pages 309–320, 2010. (Cited pages 32, 35, 37, and 38)

[FG04] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3):3–31, 2004. (Cited page 90)

[FGJK06] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. SMOQE: A system for providing secure access to XML. In *VLDB*, pages 1227–1230. ACM, 2006. (Cited pages 17 and 125)

[FGJK07] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, pages 666–675, 2007. (Cited pages iii, vi, 5, 7, 16, 17, 21, 22, 124, 128, 129, 233, and 234)

[FGK03] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *LICS*, pages 188–, 2003. (Cited page 100)

[FGM$^+$07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007. (Cited page 36)

[FGRS11] Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. Streamability of nested word transductions. In *FSTTCS*, pages 312–324, 2011. (Cited pages 40 and 102)

[FGZ12] Wenfei Fan, Floris Geerts, and Lixiao Zheng. View determinacy for preserving selected information in data transformations. *Inf. Syst.*, 37(1):1–12, 2012. (Cited page 24)

[FKSV08] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X*, 2008. (Cited pages 28 and 29)

[FM71] Michael J. Fischer and Albert R. Meyer. Boolean matrix multiplication and transitive closure. In *SWAT (FOCS)*, pages 129–131, 1971. (Cited page 47)

[FM04] Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In *SACMAT*, pages 61–69, 2004. (Cited page 4)

[FM07] Irini Fundulaki and Sebastian Maneth. Formalizing XML access control for update operations. In *SACMAT*, pages 169–174, 2007. (Cited page 38)

[FMP05] Béatrice Finance, Saïda Medjdoub, and Philippe Pucheral. The case for access control on XML relationships. In *BDA*, 2005. (Cited pages 18 and 19)

[FPZ09] J.N. Foster, B.C. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations Symposium*, 2009. (Cited page 36)

[FRR$^+$10] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. Properties of visibly pushdown transducers. In *MFCS*, pages 355–367, 2010. (Cited pages 101, 102, 103, and 178)

[FS11] Emmanuel Filiot and Frédéric Servais. Visibly Pushdown Transducers with Look-Ahead. Research report, March 2011. to appear at SOFSEM'12. (Cited page 181)

[FUV83] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *PODS*, pages 352–365, 1983. (Cited page 184)

[FWW97] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electr. Notes Theor. Comput. Sci.*, 9:27–37, 1997. (Cited page 73)

[GAP11] Alban Galland, Serge Abiteboul, and Neoklis Polyzotis. Web information management with access control. In *14th International Workshop on the Web and Databases (WebDB)*, May 2011. (Cited page 20)

[Gau09] Olivier Gauwin. *Flux XML, Requêtes XPath et Automates.* Phd thesis, Université des Sciences et Technologie de Lille - Lille I, September 2009. http://tel.archives-ouvertes.fr/tel-00421911/en/. (Cited pages 60, 62, 63, 65, 69, 74, 77, and 79)

[GD72] G. Scott Graham and Peter J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM. (Cited page 2)

[GGH$^+$09] Parke Godfrey, Jarek Gryz, Andrzej Hoppe, Wenbin Ma, and Calisto Zuzarte. Query rewrites with views for XML in DB2. In *ICDE*, pages 1339–1350, 2009. (Cited page 24)

[GGM12] Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012. (Cited page 217)

[GH08] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In *ICALP (2)*, pages 39–50, 2008. (Cited pages 54 and 293)

[GHK07] Hermann Gruber, Markus Holzer, and Martin Kutrib. The size of Higman-Haines sets. *Theor. Comput. Sci.*, 387(2):167–176, 2007. (Cited page 224)

[GHK09] Hermann Gruber, Markus Holzer, and Martin Kutrib. More on the size of Higman-Haines sets: Effective constructions. *Fundam. Inform.*, 91(1):105–121, 2009. (Cited page 224)

[GI81] E.M. Gurari and O.H Ibarra. The complexity of decision problems for finite-turn multicounter machines. *J. Comput. Syst. Sci.*, 22(2):220–229, 1981. (Cited pages 148 and 178)

[GI83] E.M. Gurari and O.H Ibarra. A note on finitely-valued and finitely ambiguous transducers. *Mathematical Systems Theory*, 16(1):61–66, 1983. (Cited page 148)

[GI08] Françoise Gire and Hicham Idabal. Updates and views dependencies in semi-structured databases. In *IDEAS*, pages 159–168, 2008. (Cited pages 29 and 30)

[GI10] Françoise Gire and Hicham Idabal. Regular tree patterns: a uniform formalism for update queries and functional dependencies in XML. In *EDBT/ICDT Workshops*, 2010. (Cited page 30)

[GIMN10] Wouter Gelade, Tomasz Idziaszek, Wim Martens, and Frank Neven. Simplifying XML schema: single-type approximations of regular tree languages. In *PODS*, pages 251–260, 2010. (Cited page 42)

[GJ08] Hermann Gruber and Jan Johannsen. Optimal lower bounds on regular expression size using communication complexity. In *FoSSaCS*, pages 273–286, 2008. (Cited pages 228 and 293)

[GK02] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *LICS*, pages 189–202, 2002. (Cited pages 46 and 90)

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190, 2003. (Cited page 90)

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005. (Cited page 92)

[Glu61] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961. (Cited page 55)

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995. (Cited page 28)

[GM11] Steven Grijzenhout and Maarten Marx. The quality of the XML web. In *CIKM*, pages 1719–1724, 2011. (Cited page 43)

[GMM10] Pierre Ganty, Rupak Majumdar, and Benjamin Monmege. Bounded underapproximations. In *CAV*, pages 600–614, 2010. (Cited page 43)

[GMS12] Benoît Groz, Sebastian Maneth, and Slawek Staworko. Deterministic regular expressions in linear time. In *PODS*, pages 49–60, 2012. (Cited pages 9 and 204)

[GNR08] Olivier Gauwin, Joachim Niehren, and Yves Roos. Streaming tree automata. *Inf. Process. Lett.*, 109(1):13–17, 2008. (Cited page 62)

[GPZ88] Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988. (Cited page 34)

[Gre68] Sheila A. Greibach. A note on undecidable properties of formal languages. *Mathematical Systems Theory*, 2(1):1–6, 1968. (Cited page 58)

[Gri68] Timothy V. Griffiths. The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. *J. ACM*, 15(3):409–413, 1968. (Cited pages 165 and 187)

[GRS08] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4), 2008. (Cited page 31)

[GSC$^+$09] Benoît Groz, Slawomir Staworko, Anne-Cécile Caron, Yves Roos, and Sophie Tison. XML security views revisited. In *DBPL*, pages 52–67, 2009. (Cited page 8)

[GSF06] José Fortes Gálvez, Sylvain Schmitz, and Jacques Farré. Shift-resolve parsing: Simple, unbounded lookahead, linear time. In *CIAA*, pages 253–264, 2006. (Cited page 43)

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997. (Cited pages 25 and 87)

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001. (Cited pages 22 and 23)

[Heg90] Stephen J. Hegner. Foundations of canonical update support for closed database views. In *ICDT*, pages 422–436, 1990. (Cited page 35)

[Heg04] Stephen J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Artif. Intell.*, 40(1-2):63–125, 2004. (Cited pages 33 and 35)

[Heg08] Stephen J. Hegner. Semantic bijectivity and the uniqueness of constant-complement updates in the relational context. In *SDKB*, pages 160–179, 2008. (Cited page 35)

[HHK11] Pierre-Cyrille Héam, Vincent Hugot, and Olga Kouchnarenko. Loops and overloops for tree walking automata. In *CIAA*, pages 166–177, 2011. (Cited page 67)

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, s3-2(1):326–336, January 1952. (Cited page 223)

[HM98] C. Hagenah and A. Muscholl. Computing epsilon-free NFA from regular expressions in $O(n \log^2(n))$ time. In *MFCS*, pages 277–285, 1998. (Cited pages 56 and 207)

[HP98] Xiaohan Huang and Victor Y. Pan. Fast rectangular matrix multiplication and applications. *J. Complexity*, 14(2):257–299, 1998. (Cited page 48)

[HPW11] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, pages 371–384, 2011. (Cited page 36)

[HS09] Yo-Sub Han and Kai Salomaa. Nondeterministic state complexity of nested word automata. *Theor. Comput. Sci.*, 410(30-32):2961–2971, 2009. (Cited page 66)

[HSW97] Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small epsilon-free nondeterministic finite automata. In *STACS*, pages 55–66, 1997. (Cited page 56)

[HSW01] Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small -free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001. (Cited page 56)

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. (Cited page 205)

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. (Cited pages 55, 58, 218, and 291)

[IL84] Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31:761–791, September 1984. (Cited page 26)

[JR08] Michael Johnson and Robert D. Rosebrugh. Constant complements, reversibility and universal view updates. In *AMAST*, pages 238–252, 2008. (Cited page 35)

[JR10] Florent Jacquemard and Michaël Rusinowitch. Rewrite-based verification of XML updates. In *PPDP*, pages 119–130, 2010. (Cited page 39)

[JWMR07] Ming Jiang, Ling Wang, Murali Mani, and Elke A. Rundensteiner. Updating views over recursive XML. In *EROW*, 2007. (Cited page 37)

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163, 1985. (Cited pages 33, 37, and 173)

[Kel86] Arthur M. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB*, pages 467–474, 1986. (Cited page 33)

[Kel87] Arthur M. Keller. Comments on bancilhon and spyratos' "update semantics and relational views". *ACM Trans. Database Syst.*, 12(3):521–523, 1987. (Cited page 34)

[KH00] Michiharu Kudo and Satoshi Hada. XML document security based on provisional authorization. In *ACM Conference on Computer and Communications Security*, pages 87–96, 2000. (Cited page 12)

[Kil11] P. Kilpeläinen. Checking determinism of XML schema content models in optimal time. *Inf. Syst.*, 36(3):596–617, 2011. (Cited pages vii, 203, 216, 217, and 288)

[Kle56] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J., 1956. (Cited page 54)

[KLM$^+$04] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora, and Susan Kotsovolos. Query rewrite for XML in Oracle XML DB. In *VLDB*, pages 1122–1133, 2004. (Cited page 24)

[KMM06] Yaron Kanza, Alberto O. Mendelzon, Renée J. Miller, and Zheng Zhang 0002. Authorization-transparent access control for XML under the non-truman model. In *EDBT*, pages 222–239, 2006. (Cited pages 14 and 19)

[KMR05] G. Kuper, F. Massacci, and N. Rassadko. Generalized XML security views. In *SACMAT '05: Proceedings of the tenth ACM Symposium on Access Control Models and Technologies*, pages 77–84. ACM, 2005. (Cited pages 7, 16, 17, 116, 128, and 234)

[KMR09] Gabriel M. Kuper, Fabio Massacci, and Nataliya Rassadko. Generalized XML security views. *Int. J. Inf. Sec.*, 8(3):173–203, 2009. (Cited pages 13, 16, 17, 128, 129, and 233)

[Koc03] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003. (Cited page 50)

[Koc10] Christoph Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010. (Cited page 28)

[Kop11] Eryk Kopczynski. Trees in trees: Is the incomplete information about a tree consistent? In *CSL*, pages 367–380, 2011. (Cited page 27)

[KQ07] Michiharu Kudo and Naizhen Qi. Access control policy models for XML. In *Secure Data Management in Decentralized Systems*, pages 97–126. 2007. (Cited page 12)

[KR10] Orna Kupferman and Adin Rosenberg. The blowup in translating LTL to deterministic automata. In *MoChArt*, pages 85–94, 2010. (Cited pages 112, 114, 150, and 278)

[KSV06] Yannis Kotidis, Divesh Srivastava, and Yannis Velegrakis. Updates through views: A new hope. In *ICDE*, page 2, 2006. (Cited page 35)

[KT07] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Inf. Comput.*, 205(6):890–916, 2007. (Cited pages vii, 203, 216, 217, and 288)

[KT10] Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *LICS*, pages 80–89, 2010. (Cited page 222)

[KV05] Orna Kupferman and Moshe Y. Vardi. From linear time to branching time. *ACM Trans. Comput. Log.*, 6(2):273–294, 2005. (Cited pages 112, 113, 114, and 277)

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000. (Cited pages 67, 68, and 127)

[Lam71] Butler W. Lampson. Protection. In *Princeton University*, pages 437–443, 1971. (Cited page 2)

[Lan11] Martin Lange. P-hardness of the emptiness problem for visibly pushdown languages. *Inf. Process. Lett.*, 111(7):338–341, 2011. (Cited page 79)

[Lec03] Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In *PODS*, pages 49–55, 2003. (Cited page 33)

[LHT07] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 21–30, 2007. (Cited page 36)

[Lib11] Leonid Libkin. Incomplete information and certain answers in general data models. In *PODS*, pages 59–70, 2011. (Cited page 27)

[Lin11] Andrzej Lingas. A fast output-sensitive algorithm for boolean matrix multiplication. *Algorithmica*, 61(1):36–50, 2011. (Cited page 48)

[LLLL11] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. QFilter: rewriting insecure XML queries to secure ones using non-deterministic finite automata. *VLDB J.*, 20(3):397–415, 2011. (Cited pages 12, 15, and 234)

[Loh10] Markus Lohrey. Compressed membership problems for regular expressions and hierarchical automata. *Int. J. Found. Comput. Sci.*, 21(5):817–841, 2010. (Cited page 147)

[LS08] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with temporal logics and automata. In *LPAR*, pages 97–112, 2008. (Cited pages 100 and 112)

[LS10] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8(2):210–232, 2010. (Cited pages 27, 100, 111, 112, 119, 161, 162, and 234)

[LV03] Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, 2003. (Cited page 34)

[Man01] Murali Mani. keeping chess alive do we need 1-unambiguous content models? In *Extreme Markup Languages*, 2001. (Cited pages ix and 236)

[Mar04] Maarten Marx. XPath with conditional axis relations. In *EDBT*, pages 477–494, 2004. (Cited pages 90, 92, 93, 94, and 127)

[Mar05a] Maarten Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005. (Cited page 93)

[Mar05b] Maarten Marx. First order paths in ordered trees. In *ICDT*, pages 114–128, 2005. (Cited page 93)

[Mar07] Maarten Marx. Queries determined by views: pack your views. In *PODS*, pages 23–30, 2007. (Cited pages 22 and 23)

[MdR05] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2):41–46, 2005. (Cited page 93)

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. (Cited page 53)

[Mét88] Yves Métivier. On recognizable subsets of free partially commutative monoids. *Theor. Comput. Sci.*, 58:201–208, 1988. (Cited page 191)

[MKSW06] Sriram Mohan, Jonathan Klinginsmith, Arijit Sengupta, and Yuqing Wu. Acxess - access control for XML with enhanced security specifications. In *ICDE*, page 171, 2006. (Cited page 19)

[MKVZ11] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, pages 972–983, 2011. (Cited page 25)

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005. (Cited pages 84, 85, and 86)

[MN00] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation, 2000. (Cited page 43)

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006. (Cited pages 44, 86, 202, and 235)

[Moh12] Mehryar Mohri. A disambiguation algorithms for finite automata and functional transducers. In *CIAA, to appear*, 2012. (Cited page 40)

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., 1956. (Cited page 53)

[Mos05] Tim Moses. eXtensible Access Control Markup Language TC v2.0 (XACML), February 2005. (Cited page 11)

[Mot89] Amihai Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, pages 339–347, 1989. (Cited page 14)

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysic*, 5:115–133, 1943. (Cited page 54)

[MP71] Robert McNaughton and Seymour A. Papert. *Counter-Free Automata.* M.I.T. Research Monograph no. 65. The MIT Press, November 1971. (Cited page 93)

[MS90] B. Moret and H. Shapiro. *Algorithms from P to NP.* Benjamin/Cummings, 1990. (Cited page 76)

[MS05] Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *VLDB*, pages 469–480, 2005. (Cited page 24)

[MTKH03] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. In *ACM Conference on Computer and Communications Security*, pages 73–84, 2003. (Cited page 12)

[MTKH06] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006. (Cited pages 13, 15, 119, and 234)

[Mun71] J. Ian Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971. (Cited page 47)

[Mur99] M. Murata. Hedge automata: a formal model for XML schemata. Unpublished manuscript, 1999. www.horobi.com/Projects/RELAX/Archive/hedge_nice.html. (Cited page 66)

[MY60] R. McNaughton and H. Yamada. Regular expressions and finite state graphs for automata. *IRE Trans. on Electronic Comput*, EC-9(1):38–47, 1960. (Cited pages 54 and 55)

[MZZ12] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. High-performance complex event processing over XML streams. In *SIGMOD Conference*, pages 253–264, 2012. (Cited page 80)

[NdB02] Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. *J. ACM*, 49(1):56–100, 2002. (Cited page 100)

[Ned00] Mark-Jan Nederhof. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000. (Cited page 42)

[NPTT05] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-Ary Queries by Tree Automata. In *DBPL*, pages 217–231, 2005. (Cited page 100)

[NS02] Frank Neven and Thomas Schwentick. Query automata over finite trees. *Theor. Comput. Sci.*, 275(1-2):633–674, 2002. (Cited page 99)

[NSV10] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3), 2010. (Cited pages viii, 22, 23, and 24)

[Okh10] Alexander Okhotin. On the state complexity of scattered substrings and superstrings. *Fundam. Inform.*, 99(3):325–338, 2010. (Cited pages 227 and 229)

[OS11] Alexander Okhotin and Kai Salomaa. Descriptional complexity of unambiguous nested word automata. In *LATA*, pages 414–426, 2011. (Cited pages 65 and 66)

[Par66] Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966. (Cited pages 220 and 221)

[Pas11] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *ICDT*, pages 220–231, 2011. (Cited pages viii, 22, and 23)

[Pla94] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA*, pages 460–470, 1994. (Cited pages 178 and 285)

[Pla10] Thomas Place. *Decidable Characterizations for Tree Logics.* Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, December 2010. (Cited page 94)

[PS10] Thomas Place and Luc Segoufin. Deciding definability in $\mathrm{FO}_2(<)$ (or XPath) on trees. In *LICS*, pages 253–262, 2010. (Cited page 94)

[PSZ11] François Picalausa, Frédéric Servais, and Esteban Zimányi. XEvolve: an XML schema evolution framework. In *SAC*, pages 1645–1650, 2011. (Cited page 80)

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *PODS*, pages 35–46, 2000. (Cited pages 84 and 85)

[PZC96] J.-L. Ponty, D. Ziadi, and J.-M. Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In *Workshop on Implementing Automata*, pages 109–119, 1996. (Cited pages 56, 206, and 207)

[Rab68] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Bull. Amer. Math. Soc.*, 74:1025–1029, 1968. (Cited page 50)

[Ras06] N. Rassadko. Policy classes and query rewriting algorithm for XML security views. In *20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec)*, volume 4127 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2006. (Cited page 17)

[Ras07] Nataliya Rassadko. Query rewriting algorithm evaluation for XML security views. In *Secure Data Management*, pages 64–80, 2007. (Cited pages 17, 116, 128, and 129)

[Rel01] 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html. (Cited page 87)

[RMSR04] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD Conference*, pages 551–562, 2004. (Cited page 14)

[RS01] Arnon Rosenthal and Edward Sciore. Administering permissions for distributed data: Factoring and automated inference. In *DBSec*, pages 91–104, 2001. (Cited page 14)

[RS06] Mukund Raghavachari and Oded Shmueli. Conflicting XML Updates. In *EDBT*, pages 552–569, 2006. (Cited pages 29 and 30)

[RS08] Jean-François Raskin and Frédéric Servais. Visibly Pushdown Transducers. In *ICALP (2)*, pages 386–397, 2008. (Cited page 101)

[Sak05] Jacques Sakarovitch. The language, the expression, and the (small) automaton. In *CIAA*, pages 15–30, 2005. (Cited page 54)

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. (Cited page 54)

[SAN10] CWE/SANS top 25 most dangerous software errors, available at: http://cwe.mitre.org/top25, 2010. (Cited page 4)

[SBG10] S. Staworko, I. Boneva, and B. Groz. The view update problem for XML. In *EDBT Workshops (Updates in XML)*. ACM, March 2010. (Cited pages 9 and 176)

[Sch65] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965. (Cited page 93)

[Sch75] Marcel Paul Schützenberger. Sur les relations rationnelles. In *Automata Theory and Formal Languages*, pages 209–213, 1975. (Cited page 178)

[Sch06] G. Schnitger. Regular expressions and NFAs without *epsilon*-transitions. In *STACS*, pages 432–443, 2006. (Cited page 56)

[Sch07] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. In *ICALP*, pages 692–703, 2007. (Cited page 43)

[Ser11] Frédéric Servais. *Visibly Pushdown Transducers*. Phd thesis, Université Libre de Bruxelles, September 2011. http://theses.ulb.ac.be/ETD-db/collection/available/ULBetd-09292011-142239/. (Cited page 102)

[SF02] A. Stoica and C. Farkas. Secure XML views. In *IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security*, volume 256 of *Research Directions in Data and Applications Security*, pages 133–146. Kluwer, 2002. (Cited page 16)

[SLLN09] Slawomir Staworko, Grégoire Laurence, Aurélien Lemay, and Joachim Niehren. Equivalence of deterministic nested word to word transducers. In *FCT*, pages 310–322, 2009. (Cited page 178)

[Son11] 2011. Playstation network consumer alert : http://us.playstation.com/news/consumeralerts. (Cited page 3)

[ST11] Maryam Shoaran and Alex Thomo. Evolving schemas for streaming XML. *Theor. Comput. Sci.*, 412(35):4545–4557, 2011. (Cited page 40)

[Sto74] Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, MIT, Cambridge, Massasuchets, USA, 1974. (Cited pages 90 and 95)

[STP+06] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal, Amr El Abbadi, and K. Selçuk Candan. Maintaining XPath views in loosely coupled systems. In *VLDB*, pages 583–594, 2006. (Cited pages 29 and 30)

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969. (Cited page 47)

[SW73] T. G. Szymanski and J. H. Williams. Non-canonical parsing. In *14th Annual Symposium on Foundations of Computer Science*, pages 122–129. IEEE, 1973. (Cited page 139)

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002. (Cited page 29)

[Tan09] Nguyen Van Tang. A tighter bound for the determinization of visibly pushdown automata. In *INFINITY*, pages 62–76, 2009. (Cited pages 73, 74, and 79)

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. (Cited page 218)

[TBS02] I. Tatarinov, K. Beyer, and J. Shanmugasundaram. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, 2002. (Cited page 37)

[tC06] Balder ten Cate. The expressivity of XPath with transitive closure. In *PODS*, pages 328–337, 2006. (Cited page 67)

[tCL09] Balder ten Cate and Carsten Lutz. The complexity of query containment in expressive fragments of XPath 2.0. *J. ACM*, 56(6), 2009. (Cited pages 68, 111, and 154) [tCS08] Balder ten Cate and Luc Segoufin. XPath, transitive closure logic, and nested tree walking automata. In *PODS*, pages 251–260, 2008. (Cited pages 93, 152, and 153)

[tCS10] Balder ten Cate and Luc Segoufin. Transitive closure logic, nested tree walking automata, and XPath. *J. ACM*, 57(3), 2010. (Cited pages 67 and 93)

[Tho68] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. (Cited page 54)

[TNP07] Salvatore La Torre, Margherita Napoli, and Mimmo Parente. The word problem for visibly pushdown languages described by grammars. *Formal Methods in System Design*, 31(3):265–279, 2007. (Cited page 79)

[To10a] Anthony Widjaja To. *Model Checking Infinite-State Systems: Generic and Specific Approaches*. PhD thesis, School of Informatics, University of Edinburgh, 2010. (Cited page 222)

[To10b] Anthony Widjaja To. Parikh images of regular languages: Complexity and applications. *CoRR*, abs/1002.1464, 2010. (Cited page 222)

[TVY08] Alex Thomo, Srinivasan Venkatesh, and Ying Ying Ye. Visibly pushdown transducers for approximate validation of streaming XML. In *FoIKS*, pages 219–238, 2008. (Cited pages 79 and 101)

[TW65] J. W. Thatcher and J. B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 820, 1965. (Cited page 59)

[TW68] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968. (Cited pages 59, 92, and 139)

[TW98] Denis Thérien and Thomas Wilke. Over words, two variables are as powerful as one quantifier alternation. In *STOC*, pages 234–240, 1998. (Cited page 94)

[TYÖ$^+$08] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, pages 873–882, 2008. (Cited page 25)

[Val75] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975. (Cited page 79)

[Var82] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982. (Cited page 90)

[Var98] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *ICALP*, pages 628–641, 1998. (Cited pages 108, 109, 110, and 111)

[Ver05] Roel Vercammen. Updating XML views. In *VLDB PhD Workshop 2005*, pages 6–10, 2005. (Cited page 36)

[VHP06] Roel Vercammen, Jan Hidders, and Jan Paredaens. Query Translation for XPath-Based Security Views. In *EDBT Workshops*, pages 250–263, 2006. (Cited pages vi, 21, and 125)

[VL08] Antti Valmari and Petri Lehtinen. Efficient minimization of DFAs with partial transition. In *STACS*, pages 645–656, 2008. (Cited page 218)

[vpa] The visibly pusdhown languages page. (Cited page 80)

[vul08] http://www.kb.cert.org/vuls/id/493881, 2008. (Cited page 4)

[W3C] http://lists.w3.org/Archives/Public/xmlschema-dev/2007Oct/0039.html. (Cited pages ix and 236)

[W3C09] W3C. XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, 2009. (Cited page 103)

[W3C11] W3C. XQuery Update Facility 1.0 Requirements. http://www.w3.org/TR/xquery-update-10-requirements/, 2011. (Cited page 104)

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012. (Cited page 47)

[WJRM08] Ling Wang, Ming Jiang, Elke A. Rundensteiner, and Murali Mani. An optimized two-step solution for updating XML views. In *DASFAA*, pages 19–34, 2008. (Cited page 37)

[WK95] Andreas Weber and Reinhard Klemm. Economy of description for single-valued transducers. *Inf. Comput.*, 118(2):327–340, 1995. (Cited page 40)

[WRM06] Ling Wang, Elke A. Rundensteiner, and Murali Mani. Updating XML views published over relational databases: Towards the existence of a correct update mapping. *Data and Knowledge Engineering*, 58, 2006. (Cited page 37)

[WSL$^+$07] Wendy Hui Wang, Divesh Srivastava, Laks V. S. Lakshmanan, SungRan Cho, and Sihem Amer-Yahia. Optimizing tree pattern queries over secure XML databases. In *Secure Data Management in Decentralized Systems*, pages 127–165. 2007. (Cited page 13)

[XML] http://www.w3.org/standards/xml/core. (Cited page 46)

[XML99] W3C. extensible markup language (XML) 1.0. http://www.w3.org/TR/xml/, 1999. (Cited pages 45, 46, and 85)

[XML04] XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028, 2004. (Cited page 85)

[xml09] 2009. http://www.pcworld.com/businesscenter/article/169825/xml_exploits_are_next_analyst_warns.html. (Cited page 4)

[XÖ05] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, pages 121–132, 2005. (Cited page 24)

[XPa99] W3C. XML path language (XPath) version 1.0. http://www.w3.org/TR/xpath, 1999. (Cited page 90)

[YSLJ04] Ting Yu, Divesh Srivastava, Laks V. S. Lakshmanan, and H. V. Jagadish. A compressed accessibility map for XML. *ACM Trans. Database Syst.*, 29(2):363–402, 2004. (Cited page 13)

[YZ05] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, 1(1):2–13, 2005. (Cited page 48)

[ZC11] Lixiao Zheng and Haiming Chen. Determinacy and rewriting of conjunctive queries over unary database schemas. In *SAC*, pages 1039–1044, 2011. (Cited page 23)

[ZGM98] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, pages 116–125, 1998. (Cited page 28)

[ZZSZ07] Huaxin Zhang, Ning Zhang, Kenneth Salem, and Donghui Zhuo. Compact access control labeling for efficient secure XML query evaluation. *Data Knowl. Eng.*, 60(2):326–344, 2007. (Cited page 13)

A. Appendix

Where the tree falls, there it will lie.

(Ecclesiastes, 11:3)

Pumping lemma for VPAs: a lower bound

Proposition 3.16: *For every n, there exists a VPA $\mathcal{A}_n$ with n states such that the smallest tree in $L(\mathcal{A}_n)$ has size $2^{\Omega(n^2)}$.*

Fix n an odd number. We define three symbols $\{l, m, r\}$, together with a strict order $l \leqslant m \leqslant r$. We define the VPA $\mathcal{A} = (\Sigma, Q, \Gamma, I, F, \Delta)$ as follows: $Q = \{i_\eta \mid \ i \in \{0, 1, \ldots, n\}, \eta \in \{l, m, r\}\} \cup \{q_i, q_f\}$, $\Gamma = Q, \Sigma = \{a\}$, $I = \{q_i\}$, $F = \{q_f\}$, and the transitions are defined by the following rules:

1. $q_i \xrightarrow{(\mathrm{op},a):q_i} 1_l$
2. $n_r \xrightarrow{(\mathrm{cl},a):q_i} q_f$

3. $i_\eta \xrightarrow{(\mathrm{op},a):i_\eta} (i-1)_l$ for all $i \geqslant 2, \eta \in \{l, m\}$
4. $(i-1)_r \xrightarrow{(\mathrm{cl},a):i_l} i_m$ for all $i \geqslant 1$
5. $(i-1)_r \xrightarrow{(\mathrm{cl},a):i_m} i_r$ for all $i \geqslant 1$

6. $0_l \xrightarrow{\varepsilon} 0_r$

7. $(i+2k-1)_r \xrightarrow{(\mathrm{cl},a):i_l} (i+k)_m$ for all $i \geqslant 2, k \geqslant 1$

8. $(j-k)_m \xrightarrow{(\mathrm{op},a):j_m} (j-1-2k)_l$ for all $k \geqslant 1, j \geqslant 2k+2$

9. $1_l \xrightarrow{(\mathrm{op},a):k_r} (n-2k+2)_l$ for all $k \geqslant 1$
10. $n_r \xrightarrow{(\mathrm{cl},a):k_r} (1+k)_m$ for all $k \geqslant 1$

11. $(j-k)_m \xrightarrow{(\mathrm{op},a):j_m} (n-2k+2)_l$ for all $k \geqslant 1, j = 2k+1$ such that $j \leqslant n$
12. $n_r \xrightarrow{(\mathrm{cl},a):j_m} j_l$ for all $j \leqslant n$

The ε-transition from rule 6 can be easily eliminated. The intuition behind these definitions is that we use the stack symbol to determine the state before opening the left child (state indexed by l) and the one after closing the right child (state indexed by r). In other words, for every tree $t \in L(\mathcal{A})$, every node n in t and every accepting run ρ over t, $\rho^{\uparrow}(n)$ is uniquely determined by $\rho(n)$. The state in the middle is not determined in this way, but cannot be arbitrary in a tree of minimal size, as we show below.

Claim: *there exists a mapping $f : Q^2 \to Q^2$ such that for every tree $t \in L(\mathcal{A})$, every node n in t and every accepting run ρ over t, $\rho^{\uparrow}(n) = f(\rho(t))$.*

This mapping can be obtained easily from the transition rules: $f(q_i, q_f) = (1_l, n_r)$ for instance, and for the appropriate values of i and k as exposed in rules 3,4 and 3,7: $f(i_l, i_m) = ((i-1)_l, (i-1)_r)$ and $f(i_l, (i+k)_m) = ((i-1)_l, (i+2k-1)_r) \ldots$

For every $i, j \leqslant n$ and every $\alpha, \beta \in \{l, m, r\}$, let $t(i_\alpha, j_\beta)$ denote the set of all hedges (or trees) of minimal size in $L(\mathcal{A}_{i_\alpha, j_\beta})$. We denote this minimal size by $s(i_\alpha, j_\beta)$, with the convention that $s(i_\alpha, j_\beta)$ is infinite if the corresponding set of hedges $L(\mathcal{A}_{i_\alpha, j_\beta})$ is empty. A quick inspection of Δ shows that $t(i_\alpha, j_\beta) = \varnothing$ if $i > j$ or $\alpha \not\leqslant \beta$ (P0). Fix i, j such that $1 \leqslant i \leqslant j \leqslant n$ and such that $d = j - i$ is even. For each $k \leqslant n$ let $S(i, k, j)$ denote the sum $S(i, k, j) = s(i_l, k_m) + s(k_m, j_r)$. From the property (P0) above we get immediately:

$$s(i_l, j_r) = \min_{k \in \{i, \ldots, j\}} S(i, k, j) \tag{A.1}$$

.

For every $(\alpha, \beta) \in \{(l, m), (m, r)\}$, the pair $f(i_\alpha, j_\beta)$ is of the form (i'_l, j'_r), and we get:

$$s(i_\alpha, j_\beta) = 1 + s(i'_l, j'_r) \tag{A.2}$$

Clearly, when $j = i$, $S(i, k, j)$ is finite only for $k = i = j$, in which case $S(i, i, i) = 2 * (2^i - 1)$. Let us now assume that $i < j$ and $S(i, k, j)$ is finite. A further inspection of Δ shows the following property of $S(i, k, j)$:

- if $d \neq 2k$ ($i + 2k \neq j$) then after applying equation A.2 to $s(i_l, k_m)$ and $s(k_m, j_r)$ we obtain at least one term $s(i'_l, j'_r)$ satisfying either $j' - i' > d$ or $j' = n, i' = n - d$.

- if $d = 2k$ ($i + 2k = j$) then after applying equation A.2 to $s(i_l, k_m)$ and $s(k_m, j_r)$ we obtain twice the same result: either $s((i-1)_l, (j-1)_r)$ if $i \geqslant 2$, or $s((n-(d-2))_l, n_r)$ if $i = 1$

In short: the gap between the left and right integer remains even, and it can decrease, by 2, only in the case where $d = 2k$ and $i = 1$, in which case the right integer is “reset” to n. Furthermore, if $S(i, k, j)$ is finite for some value of k, then it is so for $d = 2k$. Consequently, $S(i, k, j)$ is minimized for $d = 2k$.

We can now replace equation A.1 with

$$s(i_l, (i+2k)_r) = S(i, i+k, i+2k) \tag{A.3}$$

It follows easily that if we set $u_k = s((n-2k)_l, n_r)$ for each $k \leqslant (n-1)/2$ we obtain: $u_0 = 2 * (2^n - 1)$ and for every $k \in \{1, \ldots, (n-1)/2\}$: $u_k = 2 * (2^{n-2k} - 1) + 2 * 2^{n-2k} * u_{k-1}$. Hence

$$\begin{aligned} u_k &\geqslant 2 * 2^{n-2k} * u_{k-1} \\ &\geqslant (\prod_{j=1}^{k} 2 * 2^{n-2j}) * u_0 \\ &\geqslant 2^{(n+1)k} * (2^{-2\sum_{j=1}^{k} j}) * u_0 \\ &\geqslant 2^{(n+1)k} * (2^{-k(k+1)}) * 2 * (2^n - 1) \\ &\geqslant 2^{(n-k)(k+1)} \end{aligned}$$

For $n = 1 + 2k$ we thus obtain $u_k \geqslant 2^{(k+1)(k+1)} = 2^{\Omega(n^2)}$. This proves that any tree in $L(\mathcal{A})$ has size $2^{\Omega(n^2)}$, which concludes our proof since the number of states (and stack symbols) in $\mathcal{A}$ is clearly linear in n.

Doubly exponential blowup from LTL to DFA, from NavXPath to deterministic automata

Remark 3.7: *The language $L'_m = \{(a+b+\#)^*\#w\#(a+b+\#)^*\$w \mid w \in \{a,b\}^m\}$ can be expressed with an LTL formula of size $O(m^2)$ [KV05]. This formula does not exploit the "until" operator of LTL and can therefore be expressed as a NavXPath formula.*

$$\psi_n = [\,\mathsf{not} \Leftarrow]\ \mathsf{and}\ (\phi_1)\ \mathsf{and}\ \big(\Rightarrow^*::[\mathsf{self}::\$\ \mathsf{and}\ \phi_0\ \mathsf{and}\ \ \mathsf{not}(\Leftarrow^+::\$)]\big)$$

where ϕ_0 and ϕ_1 are as defined below:

$$\phi_0 = [\overbrace{\Rightarrow/[a\ \mathsf{or}\ b]/\Rightarrow/[a\ \mathsf{or}\ b]\ldots\Rightarrow/[a\ \mathsf{or}\ b]}^{m}/[\,\mathsf{not}\Rightarrow]]$$

$$\phi_1 = [\Rightarrow^*/[\mathsf{self}::\#\ \mathsf{and}\ \bigwedge_{i \leqslant m}(\phi(i,a)\ \mathsf{or}\ \phi(i,b))\ \mathsf{and}\ \Rightarrow^m::\#]]$$

with $\phi(i,a)$ and $\phi(i,b)$ defined as follows for every $i \leqslant m$:

$$\phi(i,a) = [\Rightarrow^i::a\ \mathsf{and}\ \Rightarrow^*/[\$\ \mathsf{and}\ \Rightarrow^i::a]]$$

$$\phi(i,b) = [\Rightarrow^i::b\ \mathsf{and}\ \Rightarrow^*/[\$\ \mathsf{and}\ \Rightarrow^i::b]]$$

Regular XPath formulae whose smallest model has size doubly exponential

Remark 4.7: *There exist (finitely) satisfiable $\mathcal{X}Reg$ formulae ϕ of size $O(m)$ whose smallest model has size 2^{2^m}.*

To obtain the formula we combine the proof in [ABD+05, BdRV01] with the technique from [KR10]. The proof in [BdRV01] uses a formula of size $O(m^2)$ which simulates the incrementation of a binary counter of n bits along a path: a path beginning at 0 and counting up to $2^m - 1$ has therefore length 2^m. The formula from [ABD+05] simply combines the formula from [BdRV01] that forces a branch of depth 2^m with a formula to enforce binary branching. We follow the same proof outline, but replace the formula from [BdRV01] with a formula of size $O(m)$ inspired by [KR10]. The resulting formula is quite similar to the formula from [KR10], which is natural since checking incrementation essentially consists in identifying the bit which is incremented, verifying that least significant bits are reset, and checking equality of the words formed by the most significant bits, whereas the formula from [KR10] essentially checks the equality of two words having n bits.

The example uses the alphabet $\{a_1, \dots, a_m, b_1, \dots, b_m, c\}$. Let t denote the unique tree satisfying the following two conditions:

1. t follows the DTD given by the single rule:
   $c \to ((a_1 \mid b_1), (a_2 \mid b_2), \dots (a_m \mid b_m), c, c) \mid a_1, \dots, a_m$

2. if we consider the a_i, b_i below the c nodes as representing a binary counter (with b_i standing for 1 at the i^{th} bit and a_i for 0, the least significant bit a_1/b_1 being the leftmost one), this counter is incremented by one at each parent node. In other words, each path from leaf to root counts from 0 $(a_1 \dots a_m)$ to $2^m - 1$ $(b_1 \dots b_m)$.

The tree t has depth 2^m and therefore size $m \times 2^{2^m}$. Nonetheless the following Regular XPath formula ψ_m of size $O(m)$ represents L_m:

$$\psi_n = \phi_{\text{root}} \;\mathsf{and}\; (\,\mathsf{not}[\Downarrow^* \;\mathsf{not}\,\psi_{\text{all}}])\;\mathsf{and}\;(\,\mathsf{not}[\Downarrow/\Downarrow/\Downarrow^*/[[\,\mathsf{not}\Leftarrow]\;\mathsf{and}\;[\,\mathsf{not}\,\phi_{\text{incr}}]]])$$

where ϕ_{root}, ϕ_{incr}, ψ_{all}, and their auxiliary formulae are as defined below:

$$\phi_{\text{root}} = [\mathsf{self}::c \;\mathsf{and}\; \bigwedge_{i \leqslant m} \Downarrow::b_i] \;\mathsf{and}\; [\,\mathsf{not}\Uparrow]$$

$$\psi_{\text{all}} = [(\mathsf{self}::c \;\mathsf{and}\; [[\Downarrow/\phi_0] \;\mathsf{or}\; [\Downarrow/\phi_1]])\;\mathsf{or}\;(\bigvee_{i \leqslant m}(\mathsf{self}::a_i \;\mathsf{or}\; \mathsf{self}::b_i)\;\mathsf{and}\;[\,\mathsf{not}\Downarrow])]$$

$$\phi_{\text{incr}} = (\phi_{\text{incr-0}}/\Rightarrow)^*/\phi_{\text{incr-1}}/\Rightarrow/(\phi_{\text{incr-2}}/\Rightarrow)^*/[\mathsf{self}::c \;\mathsf{or}\; [\,\mathsf{not}\Rightarrow]]$$

$$\phi_0 = [[\,\mathsf{not}\Leftarrow]/[a_1 \;\mathsf{or}\; b_1]/\Rightarrow/[a_2 \;\mathsf{or}\; b_2]/\dots/\Rightarrow/[a_m \;\mathsf{or}\; b_m]/\Rightarrow::c/\Rightarrow::c/[\,\mathsf{not}\Rightarrow]]$$

$$\phi_1 = [[\,\mathsf{not}\Leftarrow]/[a_1]/\Rightarrow/[a_2]/\dots/\Rightarrow/[a_m]/[\,\mathsf{not}\Rightarrow]]$$

$$\phi_{\text{incr-0}} = [\bigvee_{i \leqslant m-1} (b_i \;\mathsf{and}\; \Uparrow/\Leftarrow^*::a_i)]$$

$$\phi_{\text{incr-1}} = [\bigvee_{j \leqslant m} (a_j \;\mathsf{and}\; \Uparrow/\Leftarrow^*::b_j)]$$

$$\phi_{\text{incr-2}} = [\bigvee_{k\leqslant m}([a_k \;\mathsf{and}\; \Uparrow/\Leftarrow^*{::}a_k] \;\mathsf{or}\; [b_k \;\mathsf{and}\; \Uparrow/\Leftarrow^*{::}b_k])]$$

The formulae ϕ_{root} and $(\,\mathsf{not}[\Downarrow^* \;\mathsf{not}\, \psi_{\text{all}}])$ check that the DTD is satisfied, whereas ϕ_{incr} checks the incrementation of the counter, with $\phi_{\text{incr-1}}$ identifying the bit that is incremented from 0 (a_j) to 1 (b_j), $\phi_{\text{incr-0}}$ checking that every bit before was 1 and is reset to 0, and $\phi_{\text{incr-2}}$ checking that the remaining bits remain the same.

What fragment of Regular XPath is required in this example? The negation is necessary: we use the double negation to express that some property must be satisfied at every node. All axes are not required, though: The example can be adapted to use only the forward axes of XPath: $\Downarrow$ and $\Rightarrow$. For instance, we can use special symbols z_{root} and z_{fc} to identify the root of the tree and the first child of a node by requesting that formulae $\mathsf{not}\,\Downarrow^+{::}z_{\text{root}}$ and $\mathsf{not}[\Downarrow^*/\Rightarrow^+{::}z_{\text{fc}}]$ be satisfied. Furthermore, the formula does not use the full expressive power of XPath: the expression ϕ_{incr} clearly belongs to Conditional XPath. The construction in [ABD+05, BdRV01] has size $O(m^2)$ but it does not even require conditional axes; the PDL formula involved can be expressed in NavXPath.

Determinacy modulo isomorphism for interval-bounded views

Theorem 4.29: *Comparison $\preccurlyeq_3$ is in* Exptime *for interval-bounded query automata.*

We could first think of adapting immediately the proof of Lemma 4.21. Let (t, t') be a minimal pair of trees such that $\mathcal{V}iew\,(Q_2, t) \simeq \mathcal{V}iew\,(Q_2, t')$ but $\mathcal{V}iew\,(Q_1, t) \not\simeq \mathcal{V}iew\,(Q_1, t')$. Let ϕ denote an isomorphism between $\mathcal{V}iew\,(Q_2, t)$ and $\mathcal{V}iew\,(Q_2, t')$. Suppose there are three nodes $n_t^{\uparrow}, n_t^{\circ}, n_t^{\downarrow}$ in $Q_2(t)$, and three nodes $n_{t'}^{\uparrow}, n_{t'}^{\circ}, n_{t'}^{\downarrow}$ such that $n_t^{\uparrow}$ is an ancestor of n_t°, n_t° an ancestor of $n_t^{\downarrow}$, $\phi(n_t^{\uparrow}) = n_{t'}^{\uparrow}$, $\phi(n_t^{\circ}) = n_{t'}^{\circ}$, $\phi(n_t^{\downarrow}) = n_{t'}^{\downarrow}$, $\rho_t(n_t^{\uparrow}) = \rho_t(n_t^{\circ}) = \rho_t(n_t^{\downarrow})$ and $\rho_{t'}(n_{t'}^{\uparrow}) = \rho_{t'}(n_{t'}^{\circ}) = \rho_{t'}(n_{t'}^{\downarrow})$, where ρ_t, ρ_t' are defined similarly to ρ in Lemma 4.21. Replacing the subtrees below $n_t^{\uparrow}$ (resp. $n_{t'}^{\uparrow}$) with the subtree below n_t° (resp. $n_{t'}^{\circ}$), we preserve isomorphic views for Q_2. However, the views for Q_1 may become isomorphic. One could think that for at least one of the combinations for the pumping the views for Q_1 would remain non isomorphic. It so happens that this is not true, as illustrated in Figure A.1. In this figure, Q_2 selects all the nodes labeled with d, plus the root, and Q_1 selects all the nodes with label different from d. Clearly, $\mathcal{V}iew\,(Q_2, t) \simeq \mathcal{V}iew\,(Q_2, t')$ and $\mathcal{V}iew\,(Q_1, t) \not\simeq \mathcal{V}iew\,(Q_1, t')$. We can build the automata for Q_2 and Q_1 such that ρ_t (resp. $\rho_{t'}$) has the same value on all nodes labeled d in t (resp. t'). However, whatever combination is chosen for the pumping, the views for Q_1 become isomorphic after we replace the subtrees. For instance, if we replace the subtree below $n^{\uparrow}$ with the subtree below n° in both trees, the views obtained for Q_1 are both isomorphic to $r(c(a))$, and if we replace the subtree below n° with the subtree below $n^{\downarrow}$ in both trees, the views obtained for Q_1 are both isomorphic to $r(a, b, a)$.

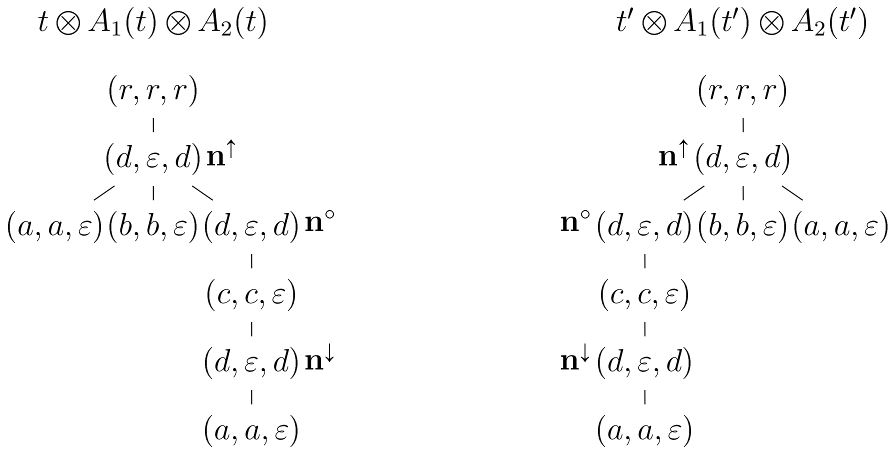

$t \otimes A_1(t) \otimes A_2(t)$
(r, r, r)
$(d, \varepsilon, d)\, \mathbf{n}^{\uparrow}$
$(a, a, \varepsilon)\, (b, b, \varepsilon)\, (d, \varepsilon, d)\, \mathbf{n}^{\circ}$
(c, c, ε)
$(d, \varepsilon, d)\, \mathbf{n}^{\downarrow}$
(a, a, ε)
$t' \otimes A_1(t') \otimes A_2(t')$
(r, r, r)
$\mathbf{n}^{\uparrow}\, (d, \varepsilon, d)$
$\mathbf{n}^{\circ}\, (d, \varepsilon, d)\, (b, b, \varepsilon)\, (a, a, \varepsilon)$
(c, c, ε)
$\mathbf{n}^{\downarrow}\, (d, \varepsilon, d)$
(a, a, ε)

Figure A.1.: The pumping of Lemma 4.21 does not work for $\preccurlyeq_3$

This suggests there is no trivial adaptation from the pumping Lemma and proof of Lemma 4.21. We therefore developed a new method, based on alignment of trees, which we discuss hereunder. We recall that $\Sigma_{\text{edit}} = \Sigma^2 \cup (\Sigma \times \{\varepsilon\}) \cup (\{\varepsilon\} \times \Sigma)$, and define the alphabet $\Sigma_{\boxtimes}$ by $\Sigma_{\boxtimes} = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ with

$$\begin{aligned}
\Sigma_1 &= \Sigma \times (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})\,, \\
\Sigma_2 &= (\{\varepsilon\} \times \{\varepsilon\} \times \Sigma) \cup (\{\varepsilon\} \times \Sigma \times \{\varepsilon\})\,, \\
\Sigma_3 &= (\{\varepsilon\} \times \{\varepsilon\} \times \Sigma \times \{op, cl\}) \cup (\{\varepsilon\} \times \Sigma \times \{\varepsilon\} \times \{op, cl\})\,.
\end{aligned}$$

Given two trees t_1, t_2 over Σ_{edit}, we denote by $t_1 \boxtimes t_2$ the *square* of t_1 and t_2, i.e., the tree over $\Sigma_{\boxtimes}$ defined by the recursive algorithm hereunder. The tree $t_1 \boxtimes t_2$ belongs to $\pi_{2,1,3}(t_1^{-1} \bowtie t_2)$ when t_1 and t_2 are upward-closed alignments, but in general t_1 and t_2 are arbitrary 2 alignments. The complexity of our definition for $t_1 \boxtimes t_2$ is explained by the necessity to handle deletions and insertions of internal nodes. One has to combine insertions of t_1 and t_2 that cover different children of a node n. This cannot be achieved using tree alignments and therefore we represent explicitly the location of the opening and closing tags of those "conflicting" nodes by special nodes labeled in Σ_3. The location of those special nodes among the children of n indicates the relationships between these nodes in t_1 and t_2, and therefore one can recover t_1 and t_2 from $t_1 \boxtimes t_2$, as we prove in Proposition A.1.
Caveat: In this whole proof, we consider trees as terms, i.e., we do not consider identifiers. Two trees will be considered equal iff they are isomorphic. We also define hedges as a sequence of trees.

A recursive definition for $t_1 \boxtimes t_2$

We define more generally operation $\boxtimes$ as a binary operation on hedges.

To prevent confusion with the symbol separating different components of a tuple, we use "·" to represents the concatenation of hedges in the definition. Similarly, we use parentheses to clarify the priority of operations, and therefore represent by $f[h]$ the tree with a root f whose children form the hedge h. The hedge $f(a(b,c),d) \cdot g$ is thus represented as $f[a[b \cdot c] \cdot d] \cdot g$. Furthermore, we note pairs/triples of symbols (i.e., tags over product alphabets like $\Sigma \times \Sigma$) between "$\langle$", "$\rangle$" instead of usual parentheses. We fix the following priorities for operations: insertion $[\,]$ of an hedge under a node has highest priority, next comes the concatenation $\cdot$ of two hedges, and $\boxtimes$ has the lowest priority. The tree $t_1 \boxtimes t_2$ is defined by the following rules. For all letters $a, b \in \Sigma$, $\alpha_1, \alpha_2 \in \Sigma_\varepsilon$, and all hedges h_1, h_2, w_1, w_2,

1. $(\langle b, \alpha_1 \rangle [h_1] \cdot w_1) \boxtimes (\langle b, \alpha_2 \rangle [h_2] \cdot w_2) = \langle b, \alpha_1, \alpha_2 \rangle [h_1 \boxtimes h_2] \cdot (w_1 \boxtimes w_2)$

2. $(\langle \varepsilon, a \rangle [h_1] \cdot w_1) \boxtimes h'$ is defined as:

   $$\begin{cases} \langle \varepsilon, a, \varepsilon \rangle [\mathcal{T}(h_1)] \cdot (w_1 \boxtimes h') & \text{if } h_1 \text{ is a hedge over } \{\varepsilon\} \times \Sigma \\ (\langle \varepsilon, a, op \rangle \cdot h_1 \cdot \langle \varepsilon, a, cl \rangle \cdot w_1) \boxtimes h' & \text{otherwise} \end{cases}$$

   where $\mathcal{T}$ is defined by $\mathcal{T}(\langle \varepsilon, c \rangle [h] \cdot w) = \langle \varepsilon, c, \varepsilon \rangle [\mathcal{T}(h)] \cdot \mathcal{T}(w)$ and the image by $\mathcal{T}$ of the empty word (neutral element of the monoid) is the empty word.

3. $(\langle \varepsilon, a, op \rangle \cdot w_1) \boxtimes h' = \langle \varepsilon, a, \varepsilon, op \rangle \cdot (w_1 \boxtimes h')$ [1]. We define symmetrically, $(\langle \varepsilon, a, cl \rangle \cdot w_1) \boxtimes h'$ as $\langle \varepsilon, a, \varepsilon, cl \rangle \cdot (w_1 \boxtimes h')$.

4. for the right operand, we add symmetrical rules to define $h \boxtimes \langle \varepsilon, a \rangle [h_2] \cdot w_2$, $h \boxtimes (\langle \varepsilon, a, op \rangle \cdot w_2)$, and $h \boxtimes (\langle \varepsilon, a, cl \rangle \cdot w_2)$. The definition of the hedge $h' \boxtimes (\langle \varepsilon, a, op \rangle \cdot w_1)$ is $\langle \varepsilon, \varepsilon, a, op \rangle \cdot (w_1 \boxtimes h')$, for instance. To keep the algorithm deterministic, we fix that rules 2 and 3 have higher priority than their right counterpart. Thus, the right rules can be applied only if no left one can.

We extend the definition to tree languages: given two $2, \Sigma$-alignment languages L_1 and L_2, we define $L_1 \boxtimes L_2$ as $\{t_1 \boxtimes t_2 \mid t_1 \in L_1, t_2 \in L_2\}$.

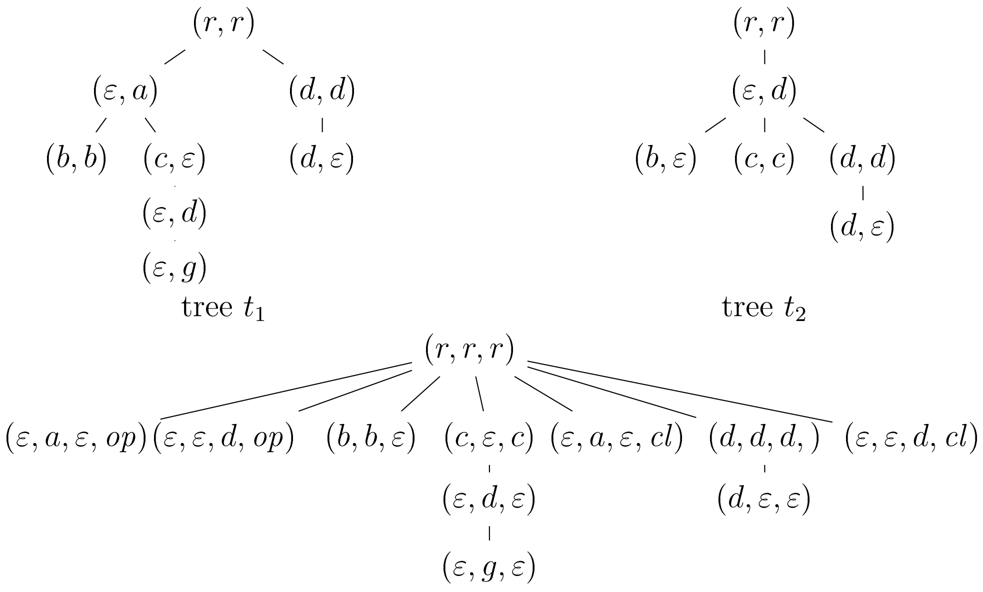
Example A.1. *In Figure A.2, we represent two alignment trees and their square.*

We define two morphisms ϕ_1, ϕ_2 on linearization of trees: ϕ_1 and ϕ_2 take as input a symbol from $\{op, cl\} \times \Sigma_\boxtimes$ and output a symbol in $\{op, cl\} \times \Sigma_{\text{edit}}$.

- $\forall \eta \in \{op, cl\}, \forall a \in \Sigma, \forall \alpha_1, \alpha_2 \in \Sigma \cup \{\varepsilon\}, \forall i \in \{1,2\}, \phi_i(\eta, a, \alpha_1, \alpha_2) = (\eta, a, \alpha_i)$ if $\alpha_i \in \Sigma$, ϵ[2] otherwise .

[1] the construction fails if a node of the form $\langle \varepsilon, a, op \rangle$ has a child

[2] the neutral element of the free monoid



tree t_1

tree t_2

tree $t = t_1 \boxtimes t_2$

Figure A.2.: Two alignment trees and their square

- $\forall \eta, \eta' \in \{op, cl\}, \forall a \in \Sigma$, $\phi_1(op, \varepsilon, a, \varepsilon, \eta') = (\eta', \varepsilon, a)$, $\phi_1(cl, \varepsilon, a, \varepsilon, \eta') = \epsilon^1$, and $\phi_1(\eta, \varepsilon, \varepsilon, a, \eta') = \epsilon^1$. Similarly, $\phi_2(op, \varepsilon, \varepsilon, a, \eta') = (\eta', \varepsilon, a)$, $\phi_2(cl, \varepsilon, \varepsilon, a, \eta') = \epsilon^1$, and $\phi_2(\eta, \varepsilon, a, \varepsilon, \eta') = \epsilon^1$.
- $\forall \eta \in \{op, cl\}, \forall a \in \Sigma$, $\phi_1(\eta, \varepsilon, a, \varepsilon) = (\eta, \varepsilon, a)$, and $\phi_1(\eta, \varepsilon, \varepsilon, a) = \epsilon^1$.

  Similarly, $\phi_2(\eta, \varepsilon, \varepsilon, a) = (\eta, \varepsilon, a)$, and $\phi_2(\eta, \varepsilon, a, \varepsilon) = \epsilon^1$.

Proposition A.1. *For every two trees t_1 and t_2 over Σ_{edit}, $t_1 \boxtimes t_2$ exists iff $\pi_1(t_1) = \pi_1(t_2)$, in which case it is a unique tree, $t_1 = \phi_1(t_1 \boxtimes t_2)$ and $t_2 = \phi_2(t_1 \boxtimes t_2)$. More accurately, we have $lin(t_1) = \phi_1(t_1 \boxtimes t_2)$.*

Proof. $t_1 \boxtimes t_2$ exists iff $\pi_1(t_1) = \pi_1(t_2)$ (recall that in this proof equality stands for isomorphism) because rule 1 is the only rule that allows a tag in Σ on the first component, and this rule requires that the same letter b occurs at the same position in $\pi_1(t_1)$ and $\pi_1(t_2)$. Clearly, this is also a sufficient condition for the existence of $t_1 \boxtimes t_2$. The priority rules make the algorithm deterministic: only one rule can be applied at any time, which guarantees the uniqueness. As for $t_1 = \phi_1(t_1 \boxtimes t_2)$ and $t_2 = \phi_2(t_1 \boxtimes t_2)$, it can be proved by induction, analysing each of the rules. □

Given two (interval bounded) queries Q_2 and Q_1 over Σ, we denote by $V_{2\to 1}(Q_2, Q_1)$ the function that maps each tree t over Σ to the tree $t' = \pi_{2,3}(t \otimes Q_2 \otimes Q_1)$. This definition is extended to languages by $V_{2\to 1}(L) = \bigcup_{t \in L} V_{2\to 1}(t)$.

Proposition A.2. *Given two k-interval bounded root preserving queries Q_1 and Q_2 with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, there is a polynomial p_0 such that one can compute an automaton $\mathcal{B}$ that accepts $V_{2\to1}(D)$ in time $(|A_{Q_1}| + |A_{Q_2}|)^{p_0(k)}$*

Proof. We first build an automaton $\mathcal{B}_0$ that accepts $L(\mathcal{B}_0) = \{t \otimes Q_1 \otimes Q_2 \mid t \in L(D)\}$ in polynomial time. $\mathcal{B}$ is built from $\mathcal{B}_0$ by projecting out the first component, with a construction similar to the one for Proposition 4.2. □

Remark A.1. *Due to the k-interval boundedness of A_2, $V_{2\to1}(D)$ presents the following property: for every t in $V_{2\to1}(D)$, for every nodes $n_1, n_2, \ldots, n_{k+1} \in N_t$, with $(n_1, n_2) \in child_t$, $(n_2, n_3) \in child_t$ $\ldots$, and $(n_k, n_{k+1}) \in child_t$, if $lab_t(n_1) \in \{\varepsilon\} \times \Sigma$, $lab_t(n_2) \in \{\varepsilon\} \times \Sigma$, $\ldots$ and $lab_t(n_{k+1}) \in \{\varepsilon\} \times \Sigma$, then for every descendant n' of n_{k+1}, $lab_t(n') \in \{\varepsilon\} \times \Sigma$.*

Proposition A.3. *Given two k-interval bounded root preserving queries Q_1 and Q_2 with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, there is a polynomial p such that one can compute an automaton $\mathcal{B}_{align}$ that accepts $V_{2\to1}(D) \boxtimes V_{2\to1}(D)$ in time $(|A_{Q_1}| + |A_{Q_2}|)^{p(k)}$*

Proof. Actually this holds not only for $V_{2\to1}$, but also for every language presenting the property in Remark A.1. Let $\mathcal{B} = (\Sigma_{\mathrm{edit}}, Q, \Gamma, I, F, R)$ be the automaton accepting $V_{2\to1}(D)$ as in Proposition A.2. We define automaton $\mathcal{B}_{align}$ as $(\Sigma', Q', \Gamma', I', F', R')$ where:

- $\Sigma' = \Sigma_{\boxtimes}$
- $Q' = Q_{13} \cup Q_2$ where $Q_{13} = Q \times Q \times \Gamma^{\leqslant k} \times \Gamma^{\leqslant k} \times \{\top, \bot\} \times \{C_l, C_r\}$ and $Q_2 = (Q \times \{\sharp\}) \cup (\{\sharp\} \times Q)$
- $\Gamma' = \Gamma_{13} \cup \Gamma_2$ where $\Gamma_{13} = \Gamma \times \Gamma \times \Gamma^{\leqslant k} \times \Gamma^{\leqslant k} \times \{\top, \bot\}$ and $\Gamma_2 = \Gamma \cup (\Gamma \times \Gamma^{\leqslant k} \times \Gamma^{\leqslant k} \times \{\top, \bot\} \times Q)$
- $I' = \{(q_l, q_r, \varepsilon, \varepsilon, \bot) \mid q_l, q_r \in I\}$
- $F' = \{(q_l, q_r, \varepsilon, \varepsilon, \bot) \mid q_l, q_r \in F\}$
- the rules in R' are defined as follows: for all $q_l, q_r, q_l', q_r' \in Q$, all $\gamma_l, \gamma_r \in \Gamma$, all $u_l, u_r \in \Gamma^{\leqslant k}$, all $\eta \in \{\bot, \top\}$, all $C \in \{C_l, C_r\}$, all $\alpha_1, \alpha_2 \in \Sigma \cup \{\varepsilon\}$, all $\theta \in \{op, cl\}$ and all $b \in \Sigma$;
  - $(q_l, q_r, u_l, u_r, \eta, C) \xrightarrow{(op,(b,\alpha_1,\alpha_2)):(\gamma_l,\gamma_r,u_l,u_r,\bot,C_l)} (q_l', q_r', \varepsilon, \varepsilon, \bot)$ is in R' if there are rules $q_l \xrightarrow{(op,(b,\alpha_1)):\gamma_l} q_l'$ and $q_r \xrightarrow{(op,(b,\alpha_2)):\gamma_r} q_r'$ in R.
  - $(q_l, q_r, \varepsilon, \varepsilon, \bot, C) \xrightarrow{(cl,(b,\alpha_1,\alpha_2)):(\gamma_l,\gamma_r,u_l,u_r,\bot,C_l)} (q_l', q_r', u_l, u_r, \bot)$ is in R' if there are rules $q_l \xrightarrow{(cl,(b,\alpha_1)):\gamma_l} q_l'$ and $q_r \xrightarrow{(cl,(b,\alpha_2)):\gamma_r} q_r'$ in R.

- $(q_l, q_r, u_l, u_r, \eta, C_l) \xrightarrow{(op,(\varepsilon,b,\varepsilon,op))(cl,(\varepsilon,b,\varepsilon,op))} (q_l', q_r, u_l \cdot \gamma_l, u_r, \top, C_l)$ is in R' if there is a rule $q_l \xrightarrow{(op,(\varepsilon,b)):\gamma_l} q_l'$ in R and $u_l \in \Gamma^{\leqslant k-1}$.

  We use a transition that does not modify the stack and reads two symbols at a time for the sake of clarity. Actually, this does not strictly follow the syntax of VPA transitions. However, it is straightforward to introduce a few new states to simulate this behaviour with two transitions, the first transition pushing a symbol into the stack which is immediately removed by the second one.

- $(q_l, q_r, u_l \cdot \gamma_l, u_r, \bot, C_l) \xrightarrow{(op,(\varepsilon,b,\varepsilon,cl))(cl,(\varepsilon,b,\varepsilon,cl))} (q_l', q_r, u_l, u_r, \bot, C_l)$ is in R' if there is a rule $q_l \xrightarrow{(cl,(\varepsilon,b)):\gamma_l} q_l'$ in R and $u_l \in \Gamma^{\leqslant k-1}$.

- The rules for $(\varepsilon, \varepsilon, b, op)$ and $(\varepsilon, \varepsilon, b, cl)$ are symmetric, except for the C_l, C_r constraints that need to be adapted, yielding rules

  $(q_l, q_r, u_l, u_r, \eta, C) \xrightarrow{(op,(\varepsilon,\varepsilon,b,op))(cl,(\varepsilon,\varepsilon,b,op))} (q_l', q_r, u_l \cdot \gamma_l, u_r, \top, C_r)$ and

  $(q_l, q_r, u_l \cdot \gamma_l, u_r, \bot, C) \xrightarrow{(op,(\varepsilon,\varepsilon,b,cl))(cl,(\varepsilon,\varepsilon,b,cl))} (q_l', q_r, u_l, u_r, \bot, C_r)$.

- $(q_l, q_r, u_l, u_r, \eta, C_l) \xrightarrow{(op,(\varepsilon,b,\varepsilon)):(\gamma_l,u_l,u_r,\eta,q_r)} (q_l', \sharp)$ is in R' if there is a rule $q_l \xrightarrow{(op,(\varepsilon,b)):\gamma_l} q_l'$ in R.

- $(q_l, \sharp) \xrightarrow{(cl,(\varepsilon,b,\varepsilon)):(\gamma_l,u_l,u_r,\eta,q_r)} (q_l', q_r, u_l, u_r, \eta, C_l)$ is in R' if there is a rule $q_l \xrightarrow{(cl,(\varepsilon,b)):\gamma_l} q_l'$ in R.

- $(q_l, \sharp) \xrightarrow{(\theta,(\varepsilon,b,\varepsilon)):\gamma_l} (q_l', \sharp)$ is in R' if rule $q_l \xrightarrow{(\theta,(\varepsilon,b)):\gamma_l} q_l'$ is in R.

- Rules for $(\varepsilon, \varepsilon, b)$ are symmetric, using states in $\{\sharp\} \times Q$ instead of $Q \times \{\sharp\}$, and replacing C_l with C_r.

Basically, we build a product automaton, and the difficulty stems from the synchronization of the stacks. The stacks are synchronized on transitions that read a letter in Σ_1. The state and stack use words u_l, u_r to simulate the runs on letters in Σ_2. The property in Remark A.1 allows to bound by k the required size for u_l and u_r. To guarantee the uniqueness property, the definition of the $\boxtimes$ operation demands that we read a letter in Σ_1 between an opening tag $(\varepsilon, b, \varepsilon, op)$ and the corresponding closing tag $(\varepsilon, b, \varepsilon, cl)$. We use $\top$ to remember this information that one has to read a letter in Σ_1 before reading the next closing tag in Σ_3. $\bot$ is used whenever there is no such constraint. Also for uniqueness, rules 2 and 3 have higher priority than their 'right' counterpart. So, no node with label in $(\{\varepsilon\} \times \{\varepsilon\} \times \Sigma)$ or $(\{\varepsilon\} \times \{\varepsilon\} \times \Sigma \times \{op, cl\})$ can be the left sibling of a node with label in $(\{\varepsilon\} \times \Sigma \times \{\varepsilon\})$ or $\cup\, (\{\varepsilon\} \times \Sigma \times \{\varepsilon\} \times \{op, cl\})$. We use C_r to remember this information: a tag C_r in the state forbids transition labeled by $(\{\varepsilon\} \times \Sigma \times \{\varepsilon\})$ or $\cup\, (\{\varepsilon\} \times \Sigma \times \{\varepsilon\} \times \{op, cl\})$. Last, but not least, all descendants of a node of the form $(\varepsilon, \varepsilon, b)$ in $t_1 \boxtimes t_2$ have label in $\{\varepsilon\} \times \{\varepsilon\} \times \Sigma$. Therefore, we do not simulate the second part of the run in that subtree, which explains why we use a state of the form $(\sharp, q)$, using the "$\sharp$" symbol on the left so as to avoid switching to symbols of the form $\{\varepsilon\} \times \Sigma \times \{\varepsilon\}$. □

Proposition A.4. *Given two k-interval bounded root preserving queries Q_1 and Q_2 with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, $Q_1 \preccurlyeq_3 Q_2$ iff morphisms ϕ_1 and ϕ_2 are equal over $V_{2\to 1}(D) \boxtimes V_{2\to 1}(D)$, i.e., iff $\forall t \in V_{2\to 1}(D) \boxtimes V_{2\to 1}(D)$, $\phi_1(t) = \phi_2(t)$.*

Corollary A.5. *Given two k-interval bounded root preserving queries Q_1 and Q_2 with $\mathrm{dom}(Q_1) = \mathrm{dom}(Q_2) = D$, $Q_1 \preccurlyeq_3 Q_2$ can be decided in exponential time*

Proof. We use Plandowski's result [Pla94] stating that equivalence of morphisms on a context-free language is decidable in polynomial time. Using this result for morphisms ϕ_1 and ϕ_2 on $L(\mathcal{B}_{align})$ we get an algorithm that works in exponential time. □

Updates and views

Theorem 5.30: *When the set of authorized editing scripts is such that the editing scripts it induces on the views are k-synchronized, we can compute an automaton for the set of all uniform view editing scripts.*

Introduction Caveat: we only present the construction for view editing scripts that rename or delete nodes (no insertion). We explain after why the construction can be extended to k-synchronized editing scripts, i.e. editing scripts that restrict the number of insertions.

We use the *fcns* encoding of trees throughout the proof: all trees considered are represented via the *fcns* encoding. Given any tree automaton $\mathcal{A} = (Q, F, \Delta)$ and state $q \in Q$ of $\mathcal{A}$, we denote by $\mathcal{L}_{\mathcal{A},q}$ the language accepted by automaton $\mathcal{A}_q = (Q, \{q\}, \Delta)$. We write $t \xrightarrow{\mathcal{A}} q$ if $t \in \mathcal{L}_{\mathcal{A},q}$. Note that for bottom-up deterministic automata, there is at most one q such that $t \xrightarrow{\mathcal{A}} q$.

We can compute automata $\mathcal{A}_V = (\Sigma \times \Sigma_\varepsilon, Q_v, F_v, \Delta_v)$ and $\mathcal{A}_p = (\Sigma \times \Sigma_\varepsilon, Q_p, F_p, \Delta_p)$ accepting languages V (*fcns*(V), to be exact) and $L \circ V$ (its fcns encoding), where L is the set of all stable editing scripts from $\mathcal{U}_s$.

We further assume that every state q_l of the tree automata $\mathcal{A}_V$ and $\mathcal{A}_P$ is "productive", i.e. there is some tree t and an accepting run ρ on that tree such that the state q_l is assigned to some node of t in ρ (hypothesis H1), and that $\mathcal{A}_V$ is bottom-up deterministic (therefore unambiguous) (hypothesis H2).

For the following construction without insertions, we could suppose $\mathcal{A}_p$ to be also bottom up deterministic. In that case, the state set of the automaton would become $Q = \mathcal{P}(Q_v \times Q_p) \times \mathcal{P}(Q_v)$: the sets of states S_p would be singletons. We do not make this assumption so that the proof remains similar when insertions are (would be) added. Anyway, this does not make the proof much heavier.

Construction First, we define a few useful notions in order to simulate all "hidden parts".

The set of failures from $(q_v, S_p) \in Q_v \times \mathcal{P}(Q_p)$ is $\mathrm{Fail}(q_v, S_p)$; the set of all $q'_v \in Q_v$ such that there exists $a \in \Sigma$, $t \in \mathit{fcns}(T_{\Sigma\times\Sigma_\varepsilon})$ such that the following two conditions hold:

- $\exists q''_v.t \xrightarrow{\mathcal{A}_V} q''_v \wedge (a, \varepsilon)(q''_v, q_v) \to q'_v \in \Delta_v$
- $\forall q''_p$ such that $t \xrightarrow{\mathcal{A}_p} q''_p$, $\forall q_p \in S_p$. $\nexists q'_p(a, \varepsilon)(q''_p, q_p) \to q'_p \in \Delta_p$.

We denote by $\rightarrowtail$ the following relation on $Q_v \times \mathcal{P}(Q_p)$: for all $q_v, q'_v \in Q_v, S_p, S'_p \subseteq Q_p$, $(q_v, S_p) \rightarrowtail (q'_v, S'_p)$ iff $q'_v \notin \mathrm{Fail}(q_v, S_p)$ and there are some $q''_v \in Q_v, q''_p \in Q_p$, some tree $t \in \mathit{fcns}(T_{\Sigma\times\Sigma_\varepsilon})$, and $a \in \Sigma$ such that the following two conditions hold:

- $\exists q''_v.t \xrightarrow{\mathcal{A}_V} q''_v \wedge (a, \varepsilon)(q''_v, q_v) \to q'_v \in \Delta_v$
- $S'_p = \{q'_p \mid \exists q_p \in S_p, \exists q''_p.t \xrightarrow{\mathcal{A}_p} q''_p.(a, \varepsilon)(q''_p, q_p) \to q'_p \in \Delta_p)\}$.

By $\rightarrowtail^*$ we denote the reflexive transitive closure of $\rightarrowtail$.

A set $C \subseteq Q_v$ will be called *persistent* if for all $q_v \in C$, for all $q'_v, q''_v \in Q_v$, and all $a \in \Sigma$; $(a, \varepsilon)(q''_v, q_v) \to q'_v \in \Delta_v \implies q'_v \in C$.

Definition A.1. *Given a persistent set $C \subseteq Q_v$, a set $E \subseteq Q_v \times \mathcal{P}(Q_p)$, the* C-guarded extension *of E is the set:*

$$\mathcal{E}_{\restriction C}(E) = \{(q'_v, S'_p) \mid q'_v \notin C \wedge \exists (q_v, S_p) \in E.(q_v, S_p) \rightarrowtail^* (q'_v, S'_p)\}$$

Definition A.2. *Given a persistent set $C \subseteq Q_v$, a set $E \subseteq (Q_v \backslash C) \times \mathcal{P}(Q_p)$ is said to be* C-saturated *if $\mathcal{E}_{\restriction C}(E) = E$ and for every (q_v, S_p) in E, Fail$(q_v, S_p) \subseteq C$.*

We can now define automaton $\mathcal{A} = (Q, F, \delta)$

- $Q = \mathcal{P}(Q_v \times \mathcal{P}(Q_p)) \times \mathcal{P}(Q_v)$.
- $F = \mathcal{P}\left(F_v \times Fin \cup (Q_v \backslash F_v) \times \mathcal{P}(Q_p)\right) \times \mathcal{P}(Q_v \backslash F_v)$ where Fin is the set of all $S \subseteq Q_p$ such that $S \cap F_p \neq \emptyset$
- The transitions are defined as follows. We fix $(E_1, C_1) \in Q$, $(E_2, C_2) \in Q$, $a \in \Sigma, \beta \in \Sigma_\varepsilon$, $C' \subseteq Q_v$. δ contains transition $(a, \beta)(E_1, C_1)(E_2, C_2) \to (E', C')$ iff there is a set $\mathcal{G} \in \mathcal{P}(Q_v \times \mathcal{P}(Q_p))$ such that all the following conditions are satisfied :
    1. C' is persistent
    2. for every $q_v \in C_1$, for every $d \in \Sigma$, $q'_v \in Q_v$ and $q''_v \in Q_v$, if $q''_v \in C_2 \cup \{q \mid \exists S.(q, S) \in E_2\}$ and $(d, a)(q_v, q''_v) \to q'_v \in \Delta_v$, then $q'_v \in C$. And symmetrically for every $q_v \in C_2$,$d \in \Sigma$...

3. for every $(q_1, S_1) \in E_1$, every $(q_2, S_2) \in E_2$, every q'_v, d such that $(d, a)(q_1, q_2) \to q'_v \in \Delta_v$, either $q'_v \in C'$ or, posing $S_p = \{q'_p \mid \exists q_p \in S_1, q''_p \in S_2.(d, \beta)(q_p, q''_p) \to q'_p \in \Delta_p\}$; $S_p \neq \varnothing$ and $(q'_v, S_p) \in \mathcal{G}$.
4. $E' = \mathcal{E}_{\upharpoonright C'}(\mathcal{G})$ and E' is C'-saturated.

For every $(E, C) \in Q$, δ contains transition $\perp \to (E, C)$ iff there are (unique by hypothesis H2) $q_v \in Q_v, S_p \subseteq Q_p$ such that $\perp \to q_v \in \Delta_v$, $S_p = \{q_p \mid \perp \to q_p \in \Delta_p\}$, C is persistent and $E = \mathcal{E}_{\upharpoonright C}((q_v, S_p))$.

Proof of the construction We must justify that $\bigcup_{q \in Q_f} \mathcal{L}_{\mathcal{A},q}$ is the set of uniform view editing scripts.

Property 1: Fix a (binary) tree t, such that $t \xrightarrow{\mathcal{A}} (E, C)$. Then for every $t' \in V$ and $q_v \in Q_v$ such that $\pi_2(t') = \pi_1(t)$ and $t' \xrightarrow{\mathcal{A}_V} q_v$, either $q_v \in C$ or there is some $S_p \subseteq Q_p$ such that both $(q_v, S_p) \in E$ and there are some $q_p \in S_p$, $t'' \in fcns(T_{\Sigma_{\text{edit}}})$ such that $t'' \xrightarrow{\mathcal{A}_p} q_p$ and $t \in (t')^{-1} \circ t''$.

Property 2: Fix a (binary) tree t accepted by $\mathcal{A}$, such that $t \xrightarrow{\mathcal{A}} (E, C)$. Then for every $t' \in V$ and $q' \in Q_v$ such that $\pi_2(t') = \pi_1(t)$ and $t' \xrightarrow{\mathcal{A}_V} q_v$, $q_v \notin C$.

Discussion: Here we presented the construction when only deletions and relabelings are allowed on the view. “Invisible” insertions are already treated: they are dealt with in the composition $\mathcal{U}_s \circ V$. When there can be no more than k insertions without a relabeling between them, we can remember in the state q_p the insertions that have been made since the last relabeling: we just need to store into the state a $k - uple$ of states from q_p. This concludes the proof.

Testing Determinism in Presence of Numeric Occurrences

In this section we explain how the algorithm to test determinism can be extended to regular expressions with numeric occurrences. We first observe that the Kleene star is unnecessary when numeric occurrence indicators are allowed: e^* can be expressed as $e^{[0..+\infty]}$. Moreover, we can also assume that in every numeric occurrence indicator $[n..m]$ the value of m is at least 2: this is because $e^{[0..1]}$ is equivalent to $e?$, and $e^{[1..1]}$ to e. The syntax of regular expressions with numeric occurrence indicators is therefore:

$$e \coloneqq a\ (a \in \Sigma) \mid (e) \odot (e) \mid (e) + (e) \mid (e)? \mid (e)^{[n..m]}$$

with $n \leqslant m$ and $m \geqslant 2$. We again assume the presence of virtual nodes # and \$ at the beginning and end of the expression. The definition of nullability and that of the first and last sets is similar to the definition for standard expressions. We assimilates each node of the parse tree with the subexpression it represents. An iterative expression (or node) is an expression of the form $(e)^{[n..m]}$.

Kilpeläinen and Tukhanen define the notion of flexibility for numeric occurrence subexpressions of e, and show that one can compute in linear time all

$n \in N_e$ that are flexible in e. They also define the relation foll_e which in some sense adapts the *Follow* relation to expressions with numeric occurrences, taking flexibility into account. The relation depends on the global expression e, but we drop the subscript to simplify the notations, since its value is always e in the following:

Definition A.3 ([KT07]). *Let f a regular expression. The relation $\mathit{foll}_e \subseteq \mathit{Pos}(e) \times \mathit{Pos}(e)$ is defined for each subexpression f of e inductively as follows:*

1. *If $f = a$ $(a \in \Sigma)$, then $\mathit{foll}(f) = [\,]$*

2. *If $f = g?$, then $\mathit{foll}(f) = \mathit{foll}(g)$*

3. *If $f = g + H$, then $\mathit{foll}(f) = \mathit{foll}(g) \cup \mathit{foll}(H)$*

4. *If $f = g \cdot H$, then $\mathit{foll}(f) = \mathit{foll}(g) \cup \mathit{foll}(H) \cup (\mathit{Last}(g) \times \mathit{First}(H))$*

5. *If $f = g^{[m..n]}$ then*

$$\mathit{foll}(f) = \begin{cases} \mathit{foll}(g) \cup (\mathit{Last}(g) \times \mathit{First}(H)) & \text{if } f \text{ is flexible in } e \\ \mathit{foll}(g) & \text{otherwise} \end{cases}$$

The determinism of regular expressions with numeric occurrence indicators can be characterized in terms of this relation:

Proposition A.6 ([KT07]). *Let e a regular expression. e is non-deterministic if and only if there are two distinct positions $x, y \in \mathit{Pos}(e)$ such that $\mathit{lab}(x) = \mathit{lab}(y)$ and:*

1. *$(z, x), (z, y) \in \mathit{foll}(e)$ for some position $z \in \mathit{Pos}(e)$, or*

2. *$(z, x) \in \mathit{foll}(g)$, $y \in \mathit{First}(g)$ and $z \in \mathit{Last}(g)$ for position z and some subexpression of the form $f = g^{[m..n]}$ in e.*

In this proposition we essentially distinguish two situations that provide a witness for non-determinism. A third situation was actually considered in [KT07, Kil11]: when both x and y belong to $\mathit{First}(e)$ they also form a witness for non-determinism, but this situation is ruled out in our setting by the introduction of the virtual nodes # and $.

What results carry over from standard expressions?

We do not modify the definitions of *SupFirst* and *SupLast* in presence of numeric occurrences. Lemma 6.6 still holds and Lemma 6.7 can be adapted as follows: if q belongs to $\mathit{foll}(p)$ then we have the two following properties: (1) $\mathit{parent}(\mathit{SupFirst}(q)) \preccurlyeq p$ and (2) $\mathit{parent}(\mathit{SupLast}(p)) \preccurlyeq q$. Lemma 6.8, however, does not hold in presence of numeric occurrences: consider for instance the positions p and q with label b and c in $(a((b+c)^{[2..3]}))d$. Then $\mathit{SupFirst}(q)$ is non-nullable, although $q \in \mathit{foll}(p)$ and $\mathit{SupLast}(p) \preccurlyeq \mathit{parent}(\mathit{SupFirst}(q))$. The consequence of this is essentially that we have to consider more cases when testing determinism: the witness for color a of a node whose right child is non-nullable may still cause non-determinism in presence of numeric occurrences. However, we can weaken Lemma 6.8:

Lemma A.7. *Let p and q be two positions of e such that $q \in \mathit{Follow}^{\odot}(p)$. If $\mathit{SupLast}(p) \preccurlyeq \mathit{parent}(\mathit{SupFirst}(q))$ then $\mathit{SupFirst}(q)$ is nullable.*

We again observe that an expression must satisfy property (P1) to be deterministic, and henceforth assume the expression satisfies (P1) since the property can be tested in linear time. The definition of $\mathit{FirstPos}(n, a)$ and $\mathit{Witness}(n, a)$ are not modified. The definition of $\mathit{Next}(n, a)$ needs only a minor modification: star expressions are replaced by flexible iterative expressions: instead of testing $\mathit{lab}(n) = *$ at line 8 of Algorithm 2, one tests if n is a flexible expression in e. Then every deterministic regular expression again satisfies property (P2), which is tested within Algorithm 2.

Then Lemma 6.11 carries over (using *foll* instead of *Follow*). However, Proposition A.6 tells us that one must also consider non-flexible iterations in addition to the *foll* relation (case 2). We therefore define $\mathit{NextNFlex}(n, a)$ to take those into account. For every node n with color a, $\mathit{NextNFlex}(n, a)$ is defined as the lowest ancestor n' of n that satisfies the following two conditions: (1) n' is a non-flexible iterative expression and (2) there exists an a-labeled position in $\mathit{First}(n')$. We can easily compute in linear time a pointer $\mathit{NextNFlex}(n, a)$ for all n of color a.

Remark A.2. *We observe that one may have $\mathit{NextNFlex}(n, a) \neq \mathit{Null}$ and $\mathit{Next}(n, a) \neq \mathit{Null}$ simultaneously even within deterministic expressions: consider the expression $e = ((aa)^{[2.2]})a$, with p_1, p_2, p_3 denoting the a-labeled positions from left to right, and with the node n denoting the subexpression (aa), with witness p_2. Then $\mathit{NextNFlex}(n, a)$ is the parent of n, $\mathit{Next}(n, a) = p_3$ and yet e is deterministic.*

We adapt Lemma 6.12 according to Proposition A.6:

Lemma A.8. *An expression e with numeric occurrences is non-deterministic iff one of the following four conditions is satisfied: (1)* (P1) *is false, (2)* (P2) *is false, (3) there exist $a \in \Sigma$, a node $n \in N_{t_a}$ of color a, and a position q in $\{\mathit{FirstPos}(n, a), \mathit{Next}(n, a)\}$ such that $\mathit{foll}^{-1}(q) \cap \mathit{foll}^{-1}(\mathit{Witness}(n, a))$ contains at least one position. (4) there exist $a \in \Sigma$, a node $n \in N_{t_a}$ of color a such that $\mathit{Last}(\mathit{NextNFlex}(n, a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n, a))$ contains at least one position.*

Algorithm Testing Determinism

In presence of numeric occurrences, it becomes slightly harder to determine if Conditions (3) and (4) of Lemma A.8 are satisfied, because we do not have an equivalent for Lemma 6.8. We therefore define a function $\mathit{HighestFlex}\,(n, n')$

which takes as input two nodes n and n' in e such that $n' \preccurlyeq n$, and returns the highest flexible iteration n'' such that $n' \preccurlyeq n'' \preccurlyeq n$ (and $n' \neq n''$). If there is no such n'' then $\mathit{HighestFlex}\,(n, n') = \mathit{Null}$. For instance in $a(b(((c^{[0..4]}).d)^{[0..\infty]}))$, if the nodes n and n' stand for the subexpressions c and $b(((c^{[0..4]}).d)^{[0..\infty]})$, then $\mathit{HighestFlex}\,(n, n')$ is the node corresponding to the subexpression $((c^{[0..4]}).d)^{[0..\infty]}$. Using techniques from [BP11], we can preprocess the parse tree of the expression in linear time so that each query $\mathit{HighestFlex}\,(n, n')$ can be answered in constant time:

Lemma A.9. *After a linear preprocessing of the expression e, each query $\mathit{HighestFlex}\,(n, n')$ can be answered in constant time.*

Proof. We compute in a simple traversal of e a pointer from each node in e to its lowest ancestor that is a flexible iteration (or the root of the tree if there is no such ancestor). Then we compute in linear time the skeleton t_{flex} of e, i.e., the tree whose nodes are the root of e plus all nodes of e that represent a flexible iteration. The tree t_{flex} is an unranked tree. We additionally keep a pointer $LC(x)$ from each node x of t_{flex} to the last (rightmost) child of x in t_{flex}. We can view t_{flex} as a binary tree, since the *fcns* encoding B_{flex} of t_{flex} can be computed in linear time. We then index B_{flex} for *LCA* queries. As observed in Fact 9.1 of [BP11], $LCA_{B_{\text{flex}}}(LC(x), y)$ returns the child of x that is an ancestor of y in t_{flex}, for any nodes $x \preccurlyeq y$ in t_{flex}.

This allows us to compute $\mathit{HighestFlex}\,(n, n')$ in constant time: we follow the precomputed pointers to retrieve the lowest ancestors y (resp. x) of n (resp. n') that are flexible iterations. If $y = x$ then $\mathit{HighestFlex}\,(n, n') = \mathit{Null}$. Otherwise $\mathit{HighestFlex}\,(n, n')$ is obtained as $LCA_{B_{\text{flex}}}(LC(x), y)$. □

We next introduce a last notation. Let us denote by x the lowest of $\mathit{SupLast}(n)$ and $\mathit{SupFirst}(n)$. We then define $\mathit{Iter}(n, a)$ as follows. If $\mathit{HighestFlex}\,(n, x) \neq \mathit{Null}$ then $\mathit{Iter}(n, a) = \mathit{HighestFlex}\,(n, x)$. Else $\mathit{Iter}(n, a) = \mathit{NextNFlex}(n, a)$ if $\mathit{NextNFlex}(n, a) \neq \mathit{Null}$ and $\mathit{NextNFlex}(n, a)$ is a descendant of both $\mathit{SupLast}(n)$ and $\mathit{SupFirst}(n)$. Otherwise, $\mathit{Iter}(n, a) = \mathit{Null}$.

Theorem A.10. *An expression e with numeric occurrences is non-determini if and only if it does not satisfy (P1) or (P2), or there exists $a \in \Sigma$ and a node n with color a such that one of the following conditions is satisfied:*

1. *$\mathit{Next}(n, a)$, $\mathit{NextNFlex}(n, a)$ and $\mathit{Iter}(n, a)$ are not all equal to Null, and one of the following two conditions is satisfied*
   - *$\mathit{Rchild}(n)$ is nullable or*
   - *$\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{Witness}(n, a), n)) \preccurlyeq n$.*
2. *or $\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{FirstPos}(n, a), n)) \preccurlyeq \mathit{Lchild}(n)$, and $\mathit{FirstPos}$ is not equal to Null.*

Proof sketch. The theorem follows from Lemma A.8 and the following characterization:

- $\mathit{foll}^{-1}(\mathit{Next}(n,a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n,a)) \neq \varnothing$ iff $\mathit{Next}(n,a) \neq \mathit{Null}$ and one of the following two conditions is satisfied:

  A1 $\mathit{Rchild}(n)$ is nullable or

  A2 $\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{Witness}(n,a),n)) \preccurlyeq n$.

- $\mathit{Last}(\mathit{NextNFlex}(n,a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n,a)) \neq \varnothing$ iff $\mathit{NextNFlex}(n,a) \neq \mathit{Null}$ and one of the following conditions is satisfied:

  B1 $\mathit{Rchild}(n)$ is nullable

  B2 $\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{Witness}(n,a),n)) \preccurlyeq n$.

- $\mathit{foll}^{-1}(\mathit{FirstPos}(n,a)) \cap \mathit{foll}^{-1}(\mathit{Witness}(n,a)) \neq \varnothing$ iff $\mathit{FirstPos}(n,a) \neq \mathit{Null}$ and one of the following conditions is satisfied:

  C1 $\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{FirstPos}(n,a),n))$ is an ancestor of $\mathit{Lchild}(n)$ (possibly $\mathit{Lchild}(n)$ itself)

  C2 $\mathit{Iter}(n,a) \neq \mathit{Null}$ and $\mathit{Rchild}(n)$ is nullable

  C3 $\mathit{Iter}(n,a) \neq \mathit{Null}$ and $\mathit{SupLast}(\mathit{HighestFlex}\,(\mathit{Witness}(n,a),n))$ is an ancestor of n □

From this theorem we get immediately a linear algorithm to test determinism.
Theorem 6.15: *Determinism of a regular expression e with numeric occurrences can be tested in linear time $O(|e|)$, for an arbitrary alphabet.*

Testing Determinism of Regular Languages

Theorem 3 from [BGMN09]: *Given a regular expression e, the problem of deciding whether $L(e)$ is deterministic is* PSPACE*-hard*

The proof in [BGMN09] is a one-page long reduction from Corridor Tiling. We therefore give a shorter proof, along the lines of Proposition 3.4 and therefore inspired from [HU79]. In the eventuality that testing determinism might prove EXPTIME-hard (which we have no inclination to believe), the proof in [BGMN09] will stand on its own merit, because some versions of the Corridor Tiling problem are EXPTIME-hard, suggesting the possibility to adapt the proof of [BGMN09]. But this remains an open question.

Proof. Brüggeman-Klein and Wood [BKW98] show that the language of expression $(a+b)^*a(a+b)$ is not deterministic. Let e an expression over an alphabet Σ and $\#$ a symbol outside Σ. Clearly, the language of $\Sigma^*\#(a+b)^*a(a+b)+e\#\Sigma^*$ is deterministic if and only if $L(e)=\Sigma^*$, which completes the reduction. Hence the PSPACE-hardness of testing determinism, by reduction from universality of regular expressions, a problem known to be PSPACE-complete.□

The DFA representation for the subwords of a regular language cannot be polynomial in general

Lemma 6.28: *There exist a family of DFA A_n with size n such that any deterministic expression accepting the subwords of $L(A_n)$ has size $n^{\Omega(\log\log n)}$.*

We denote by A_n the DFA called the "half-complete graph" in [EZ74]. The DFA $A_n = (\Sigma, Q, i, F, \Delta)$ has states $Q = \{1, \ldots, n\}$, initial state $i = 1$, final state n, alphabet Q^2, and transitions $\Delta = \{(i,(i,j),j) \mid i,j \in Q\}$. Let e_n a regular expression accepting the subwords of $L(A_n)$.

Claim: *The size of e_n is at least $n^{\Omega(\log\log n)}$.*

Proof. We assume that expressions are given by their parse tree. We need to adapt the proof of [EZ74] because their lower bounds are obtained only for particular regular expressions (those that can be represented by sheaves). One cannot assume that a minimal expression for the subwords of $L(a_n)$ is of that particular form. However, we show that the proof still holds in our setting.

For every word $w \in L(e_n)$, one can build a parse tree of w with respect to e_n. This parse tree $P(e_n, w)$ (or simply $P(w)$) is a binary tree with internal nodes labeled by internal nodes of e_n whose label is $\odot$, and leaves labeled by positions of e_n. Formally, the parse tree $P(e,u)$ of a word u with respect to e is defined as follows, where we identify subexpressions with their corresponding node:

- if the root of e is labeled $+$ then $P(e,u)$ is defined as $P(\mathit{Lchild}(e),u)$ when $u \in L(\mathit{Lchild}(e))$, and as $P(\mathit{Rchild}(e),u)$ otherwise
- if the root of e is labeled $\odot$ then let s,t two words such that $st = u$, s is accepted by the left subexpression of e and r by the right. Then the root of $P(e,u)$ is the root of e, and its left and right subtrees are formed by the corresponding parse trees for s and t.
- if the expression consists of a single position, then the parse tree of u consists of this single position (in that case u has a single letter, which is the label of the position)

Clearly, the parse tree of a word w of length $k+1$ has $k+1$ leaves which are positions of e_n, plus k internal nodes, which are the $\odot$-labeled ancestors in e of those positions. As in [EZ74], we map each node of e_n to a state of A_n: node $x \in N_{e_n}$ is mapped to the highest natural $\sigma(x) \leqslant n$ for which there exists a position of the form $(z, \sigma(x))$ below x. We extend the mapping σ to trees: σ relabels any internal node x with $\sigma(x)$, and deletes the leaves.

The crux of the proof is the following observation: for any pair of distinct words w, w' of the form $(1,i_1)(i_1,i_2)\ldots(i_k,n)$, the trees $\sigma(P(w))$ and $\sigma(P(w'))$ (considered as terms) are distinct. That is, not only do these trees have distinct sets of nodes, but there cannot be an isomorphism (preserving the labels) between them. In other words, the term $\sigma(P(w))$ determines w. In our setting, unlike for the sheaves of [EZ74], the property would not hold if we considered arbitrary words in $L(e_n)$. The property holds for words like above because a node x with label i_j in $\sigma(P(w))$ determines the second component in the last letter of the word s matched by its left child, as well as the first component of the first letter of the word t matched by its right child. This implies by induction that $i_1, \ldots, i_k$ are all determined by $\sigma(P(w))$.

This property is sufficient to apply the remainder of the proof from [EZ74]. Following faithfully [EZ74][3], we next show that there there must be many such trees, and that this huge number of trees implies a lower bound on the size of e_n.

There are $\binom{n-2}{k}$ words of the form above, which is more than $(n-2)^k/k^k$. Therefore, $(n-2)^k/k^k$ is a lower estimate for the number of different trees $\sigma(P(w))$. We next derive for that number of trees an upper estimate which involves the size of e_n. The upper estimate is obtained by counting the number of non-isomorphic binary trees with k internal nodes over a unary alphabet, and then multiplying this number by the maximum number of ways to label those trees. There are at most $\binom{2k}{k}/(k+1)$ and for our purpose at most 4^k binary trees with k internal nodes. To obtain a bound for the number of labelings, Ehrenfeucht and Zeiger observe that this labeling is determined by the pair formed by

- the labeling of all nodes along the path from the root to some maximally deep internal node x,
- and the labeling of every node outside this path.

The number of possible labelings for the path from the root to x is at most $|e_n|$. This is because when we fix x to some node of e_n, we determine its label and the label of its ancestors: the k^{th} ancestor of x in the path is then the k^{th} $\odot$-labeled ancestor of x in e_n, and their label is then obtained from σ. Furthermore, the path from the root to x contains at least $\log(k+1)$ nodes. Therefore the number of nodes outside this path is at most $k - \log(k+1)$, hence an upper bound of $(n-2)^{k-\log(k+1)}$ for the number of ways to label those nodes. Summing up, we obtain that $(n-2)^k/k^k \leqslant 4^k \times |e_n| \times (n-2)^{k-\log(k+1)}$. Consequently, $|e_n| \geqslant (n-2)^{\log(k+1)}/(4k)^k$, which, for $k = 1/3\log(n)$, yields $|e_n| \geqslant (n-2)^{2/3(\log(1/3\log(n-2))-1)}$.

We observe that Ehrenfeucht and Zeiger obtained a larger blowup for the simpler expressions they call the "complete graph". This family cannot be used directly to improve our result because the corresponding automaton has a single strongly connected component, and therefore the subword approximation for that language is trivial. It is not yet clear to us whether further ideas from this paper or from [GJ08, GH08] can be exploited to tighten the gap. □

[3] Actually, we follow the version from the technical report (available online), where the analysis is a little more detailed